An empirical comparison between MapReduce and Spark

A thesis presented in partial
fulfilment of the requirements

for the degree of

Master of Science in Information Sciences

at Massey University, Auckland, New Zealand

YuJia Liu

2019

Contents

List of Figures

List of Tables

Abstract

Nowadays, big data has become a hot topic around the world. Thus, how to store, process and analysis this big volume of data has become a challenge to different companies. The advent of distributive computing frameworks provides one efficient solution for the problem. Among the frameworks, Hadoop and Spark are the two that widely used and accepted by the big data community. Based on that, we conduct a research to compare the performance between Hadoop and Spark and how parameters tuning can affect the results.

The main objective of our research is to understand the difference between Spark and MapReduce as well as find the ideal parameters that can improve the efficiency. In this paper, we extend a novel package called HiBench suite which provides multiple workloads to test the performance of the clusters from many aspects. Hence, we select three workloads from the package that can represent the most common application in our daily life: Wordcount (aggregation job),TeraSort (shuffle/sort job) and K-means (iterative job). Through a large number of experiments, we find that Spark is superior to Hadoop for aggreation and iterative jobs while Hadoop shows its advantages when processing the shuffle/sort jobs. Besides, we also provide many suggestions for the three workloads to improve the efficiency by parameter tuning. In the future, we are going to further our research to find out whether there are some other factors that may affect the efficiency of the jobs.

Keywords: Big Data, Spark, Hadoop, HiBench suite, parameters tuning

Acknowledgements

I would like to express my thanks and gratitude to my supervisors Dr Andre Barczak and Dr Teo Susnjak for their guidance and support before,during and after my research. Thanks for giving me the opportunity to study on the big data. Thanks for spending time with me talking about the project. Thanks for their ideas and suggestions that make my research better.

Finally, I would like to thanks Massey university and all the staff that provide me a comfortable and convenient environment during my research period. Many thanks.

Chapter 1

Introduction

Nowadays, large scale of data is generated by different users around the world. These data is under different formats and most of them are unstructured. In addition, the advent of many new technologies bring much larger volumes of complex data which includes social media data, machine data and system data [1]. Thus, how to process and analyse the growing data becomes a challenge to many companies. To handle the problem, distributed computing is proposed and becomes the most efficient and fault-tolerant method for companies to save and process the massive data. Among this new group, Hadoop and Spark are the most commonly used cluster computing tools that provide the users various functions with simple API.

Hadoop is an open source cluster computing framework based on Java [2]. It is a programming model that provides users with analysis and storage infrastructure [1]. Its cluster includes one master node to assign and monitor tasks and multiple data nodes to save and conduct parallel computing. Hadoop adopts master/slave architecture which means the master node will manage all the data nodes when processing large scale of data. The cores parts for Hadoop are: MapReduce and HDFS[1] [3]. HDFS is distributed file system while MapReduce is a framework used for distributed computing. Hadoop is a combination of these two.

Spark is designed based on the Hadoop cluster and its purpose is building a program model that " fits wider class of applications than MapReduce while maintaining the automatic fault-tolerance" [3]. It is not only an alternative to the Hadoop framework but also provides various functions to process the real streaming data. Apart from map and reduce functions, Spark also support *MLib*[2],*GraphX* and *Spark streaming* for big data analysis. Spark chooses *Scala* as its default language and provides the interfaces in *Java* and *Python* to enhance its scalability. There are two important terms proposed by Spark: directed acyclic graph (DAG) and Resilient Distributed Datasets(RDD).

[1]The system to store big volume of data.
[2]Machine learning library.

The research included in this thesis conduct experiments that are modified from the HiBench[3], to test the performance of our cluster, facilitate the users to monitor the jobs as well as give the users enough freedom to modify the parameters they want. These new experiments make up the drawbacks of different cluster performance application, by implementing the jobs on our own cluster and simplifying the complex process of processing large volume of data.

These new experiments are applied to test the efficiency of different jobs under Hadoop and Spark, where the datasets are repeatedly changing and the different parameters needs to be set to ensure the job efficiency. The parameters problem is one case that multiple aspects are required to enhance the job efficiency. To make sure the experiment results are convincing,each experiment is tested for 5 times to get the average execution time. Also, we select several parameters from different aspects suggested by MapReduce and Spark online documents to enhance the correlation to the jobs. All the details of the experiments can be seen from Ambari UI[4].

1.1 Research Objectives

The primary objective of this research is to understand how Spark and MapReduce process different kinds of jobs, providing them the guide to improve the job efficiency by tuning parameters from several sides. These new experiments should allow the users to check any execution results they want and view all the job settings, they will give them a clear vision of the situation of the jobs and help them make improvements easily. To test the capabilities of our cluster, different types and sizes of the datasets need to be created and executed. Through our experiments, we need to present the difference between MapReduce and Spark as well as clearly show how parameters affect the efficiency. The objectives of this research are shown below:

1. Propose a method to extend the HiBench workloads to our own cluster.
2. Select three workloads from the HiBench and define them as different types of jobs.
3. Find the related parameters for each type of job and configure the jobs with different settings.
4. Compare Hadoop and Spark for the performance difference and how the parameters tuning affect the results.
5. Find the ideal parameters that have positive effects on Hadoop and Spark jobs.
6. Give the suggestions to different types of jobs based on our findings.

[3]An open-source package from github to test the performance of clusters.
[4]The web UI that provides the intuitive and user-friendly function to manage the Hadoop cluster.

1.2 Significance of our research

This research investigates three types of jobs from the HiBench which are defined as workloads and shows a modified implementation of the workloads on Spark and Hadoop. Also, these experiments put parameters tuning and results monitoring into consideration and thus include all the process of big data processing. Besides, the different sizes of datasets can simulate the situation of the real-world requirement and therefore provides us an idea about how to solve our daily problems with Hadoop and Spark.

This research is based on the recent research undertaken by Yassir Samadi [4],he ran all the HiBench workloads with the default settings. Since HiBench only provides several parameters under its configuration files and there are only four values to visualize the performance, we decide to extend it to control several workloads that includes configure our own settings and monitor the results from different aspects. Our research extends Yassir Samadi's research by transferring the workloads to our own cluster and building a complete process of the jobs for users to test the individual cluster performance. Based on the research,we can see the difference between Hadoop and Spark jobs clearly and how parameters can affect the results.

1.3 Scope and limitations

This research is based on the HiBench and thus is applicable to any clusters. However, since there are some modifications from the source code and configuration files, the users need to re-compress the jar package and use the command provided on the chapter 4 to execute the workloads.

The limitations of our research are shown below:

1. The input data is created randomly and may cause some deviation compared to the real-world situation.

2. The size of our cluster is small containing 10 nodes with 420GB memory and 250 vcores, compared to other's research.

3. There is strong randomness for our jobs caused by some unavoidable factors like internet problems or hardware restriction which means the execution time of the same job fluctuates strongly.

4. Some other factors like CPU consumption or I/O ratio that express the usage of the cluster resources can be added to evaluate the jobs comprehensively.

5. Only resource utilization, input splits and map/reduce side parameters are considered into our research.

1.4 Structure of the Thesis

This thesis begins with an introduction of two relevant distributive computation frameworks: Hadoop and Spark, both of them are widely used for processing big data. Next, there is a detailed review on the HiBench package and three types of workloads used in this thesis. Then, the workflow and the settings of our experiments are discussed that include the software and hardware settings. On the following chapter is the methodology which explains the ideas of setting our experiments. Chapter 5 presents the experiment results and analysis while Chapter 6 summarise our findings and provide the suggestions based on our experience. Finally, Chapter 7 concludes our research and ends our thesis with the future work.

Chapter 2

Literature Review

This chapter covers a brief explanation of Hadoop and Spark; how these frameworks work; Hibench suite; several workloads inside and the strategies for optimization Spark and Hadoop jobs. Firstly, we are going to introduce the core parts of Hadoop and Spark and how these two frameworks work. Then, we explain a new term: Hibench which is a package designed by Intel to test the performance of the Cluster. After a brief explanation of the HiBench, we are going to have a discuss about this drawbacks and pick several workloads to study further. Finally, we are going to explain something related to parameters tuning to improve the efficiency of the workloads and make the conclusion for the chapter.

2.1 Hadoop

For Hadoop, the core parts are MapReduce and HDFS. The former is a computing principle and the latter is a storage platform for big data processing.

HDFS:

It is impossible to save or process large volumes of data on one single node, therefore Hadoop proposes a distributed file management system called HDFS [5] which splits the files into small pieces(blocks) and saves them on different nodes. There are two kinds of nodes on HDFS: data node (worker) and master (name) node [6]. All the operations including delete, read and write are based on these two. The workflow of HDFS is like that: Firstly, the name node ask for access permission. If accepted, the master node will turn the file name into a HDFS block ID list which includes the file and the data notes that save the blocks related to the file. Then the ID list will be sent back to client and the users can do the further operation based on that.

MapReduce:

MapReduce is a computing framework which includes two operation: Mapper and Reducer.

Mapper:

After reading the files from HDFS, Hadoop will parse each line as a key-value pair and call the map function for each key-value pair [7]. The mapper will process them based on the map function and transfer them into the new key-value pairs [8]. Next, the new key-values pairs will be assigned to different partitions and will be sorted based on their keys. Combiner is optional and can be recognized as a local reduce operation which allows counting the values with same key in advance to reduce the I/O pressure. Finally, partitions will divide the intermediate key-value pairs into different pieces and transfer them to reducer.

Reducer:

Before conducting reduce process, MapReduce needs to implement one operation: shuffle which means transferring the mapper output data to the proper reducer. After shuffle process finished, the reducer will start some copy threads (Fetcher) and obtain the output file of the map task through HTTP [9]. Next step is about merging the outputs into different final files that will be recognized as the input data of reducer. After that, the reducer processes the data based on the reduce function and write the outputs back to the HDFS.

2.1.1 How Hadoop MapReduce works

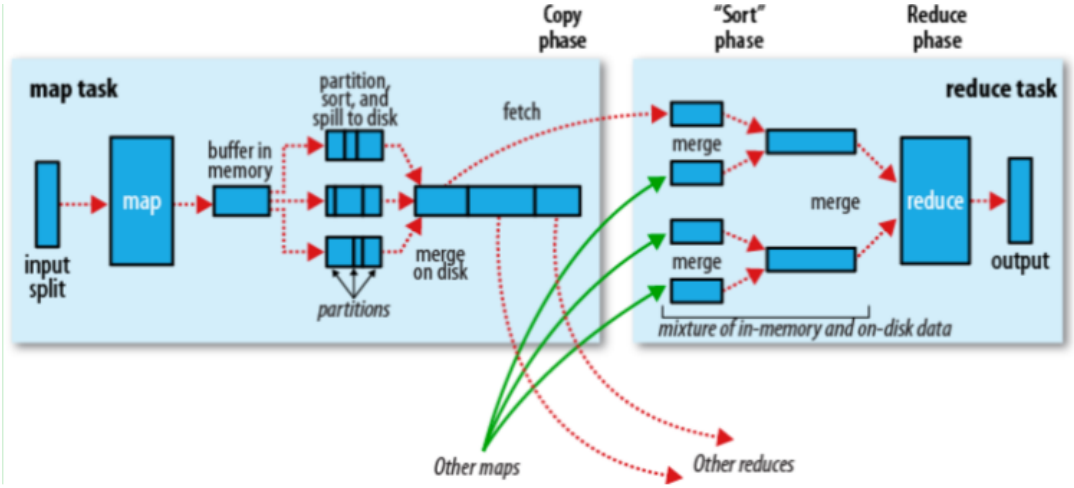

Figure 2.1: MapReduce Workflow from [10]

Figure 2.1 illustrates the workflow of MapReduce. From the figure, we can find that MapReduce job contains two side tasks which are map side tasks and reduce side tasks and each of them includes several stages. Next, we are going to explain each stage to present how the framework works.

For Map side tasks, it experiences five stages: input splits, Map function, buffer, spill and merge [11] [12]. Before conducting map tasks, the input files will be split into many pieces and each of them will be matched to one map task. The input splits are not used to store the data. On the contrary, it is an array which records the length of the slice and the position of the data. The input split is often closely related to the block of the HDFS and the default value is 128MB [13].

After receiving the input files, MapReduce will process the data based on the map function and the output will be saved on a memory buffer temporarily. Once the buffer meets the threshold, the buffer will spill to a new file and the data in the buffer will be written to this file. In addition, before writing the file into disk, MapReduce will split the data into different partitions based on the number of reduce tasks to make sure each reduce tasks correspond to one partition.

When the map function finishes, there are many spill files saved on the local file systems. Thus, MapReduce will merge these files together. Also, the sort and combiner operation will be conducted during the merging process aiming at minimizing the amount of data written back to HDFS for each time as well as reducing the amount of data transmitted to the next phase through the internet.

The last step of map task is copying the data from different partitions to the corresponding reduce tasks. This stage represents the process to transferring the output of map tasks to the reduce tasks. Also, it can be recognized as the key factor of our optimization.

For the reduce tasks, there are three stages during the process which are merging, reduce function and output. Since reduce tasks will receive different output files from map tasks and all the files are in order, it will save these files based on their sizes. If the sizes are small, reduce tasks will cache them into memory for further processing while it will merge them together and spill to the local file system if the sizes are huge.

With the number of spill files increases, MapReduce will merge them and use them as input data for reduce function. Then, after reduce function finished, the output files will be saved back to HDFS.

Although MapReduce is attractive for users because of its simplicity and user-friendly, the framework still has some limitations [14]. In MapReduce, every job needs to read the input data, process and write the results back to the HDFS [15]. It has to repeat the cycle many times when the new job requires the results from the previous job. Thus, the efficiency of MapReduce jobs is low and much resources is wasted during the process when processing the iterative tasks. To overcome the limitations of MapReduce, Spark is being

proposed.

2.2 Spark

There are two important terms proposed by Spark: RDD and DAG. These two work together perfectly and accelerate Spark 10 times as fast as Hadoop under some certain circumstances.

RDD:

Resilient Distributed Datasets (RDDs) is "an abstraction for a collection of data that can be stored and processed in memory" [16]. It is a special collection which supports multiple sources, has a fault tolerance mechanism, can be cached and supports parallel operations. Also, it can represent one dataset with multiple partitions. When running on the Hadoop cluster, RDDs will be created on the HDFS in many formats supported by Hadoop like text, sequence files. There are two kinds of operations supported by RDD: transformation and action.

Transformation:

Transformation operation adopts delay calculation which means creating a new dataset based on an existing dataset [17]. When an RDD is converted to another, there is no immediate conversion. On the contrary, it just records the logical operation of the data set for further processing. All the transformation operations are lazy which means they will not compute the result right away. The transformations will be computed only when an action operation requires a result from the driver program. The normal transformation operation includes *map, flatMap, filter, groupbykey and reduceByKey* [18].

Action:

Action operation will launch Spark jobs and return the results back to the driver program after computing process finishes. It will be triggered when the user needs to return the computing results back to the driver program or write the result to the external system. When the action operation happenes, the DAG scheduler will be trigged to divide the DAG into different stages. Then, after the process is completed, TaskScheduler is called to distribute the task to different executors. The action operation includes *count, collect* and *reduce* [18].

DAG:

Spark provides advanced DAG scheduler [19] system to express the dependencies of RDDs. Each Spark job will create a DAG and the scheduler will dive the graph into different stages

of tasks. Then the tasks will be launched to the cluster. DAG will be created in both map and reduce stages to fully express the dependencies. Also, this setting helps the simple job finishes within one stage while the complex job finishes into several stages other than splitting into different jobs. That is one reason why Spark is faster than Hadoop.

2.2.1 How Spark works

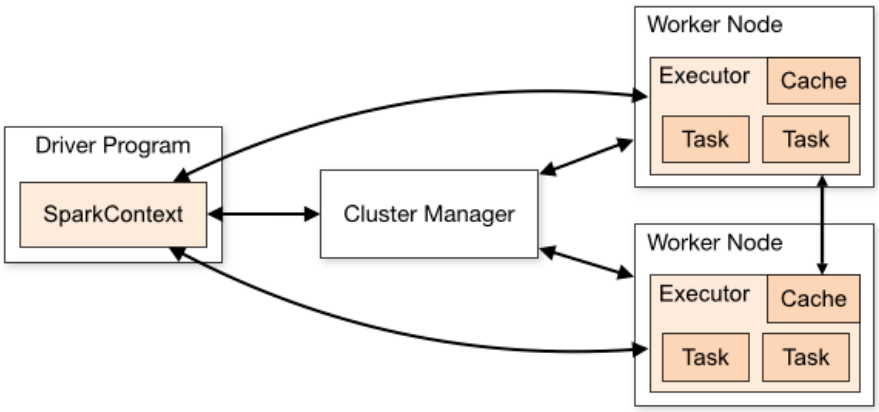

Figure 2.2: Spark Workflow from [20]

Figure 2.2 illustrates the workflow of Spark. The figure shows that Spark job will launch a corresponding driver program when the user submit a spark job through *spark-submit* command. Depends on the deploy-mode, the driver program may launch locally or on one working node of the cluster. Also, the driver process will take some resources based on the settings to control all the jobs. The first task the driver program needs to do is applying the resource which is represented by the executor process from the resource manager. There are several resource manager for users to choose from: *standalone*, *Yarn* and *Mesos* [21]. Then, the resource manager will launch several executors on the working nodes based on the user settings and each of them owns a number of CPU cores and memory.

After applying enough resource from the resource manager, driver program will start scheduling and executing our codes. Firstly, Spark will split the code into several stages. Each stage will execute a piece of code and creating a batch of tasks for the stage. Then, the tasks will be assinged to different executors to execute. Tasks are the smallest unit of Spark jobs and are responsible for the same calculation logic or code while processing different datasets. The job will move to the next stage when all the tasks of one stage finished and the results will be written to the local file system on different nodes. The next stage will utilize the result of the last stage as the input data and repeat the cycle until all the stage finishes.

Spark jobs divide different stages based on the shuffle or action operation. Once the codes includes any action operation like *reduceByKey* or *join*, Spark will divide a stage boundary there and split it as two stages. In other words, the codes before action operation and the codes after the action operation will be recognized as two stages. Thus, each task of stage may pull the data it needs from the previous stage through the Internet and execute the operation based on the codes. This process is called shuffle in Spark.

Besides, Spark provides the function to save the data into memory if the users set the *persist/cache* operation [22]. This is the main difference between Spark and MapReduce frameworks. There are three persist levels provided by Spark: MEMORY_ONLY,DISK_ONLY and MEMORY_AND_DISK. By setting different levels of persist strategies, the results from different tasks can be saved on the memory of the executor or the local file system.

The executor memory contains three parts [23]: one is utilized for tasks to execute codes which takes 20% of the total memory by default. The second part is utilized for the tasks to pull the output of the previous stage through the shuffle process and use it for aggregation and other operations. The default settings for the second part also takes 20% of the total memory. The third part is used for caching the RDDs into memory which takes 60% of the total memory based on the default settings.

2.3 HiBench Suite

The emergence of Hadoop and Spark makes it possible to save and process large volumes of data. Then how to quantitatively evaluate the performance of the clusters becomes a serious problem. There are some existing benchmarks which provide some examples to evaluate the performance like YSCB and CloudSuite [24]. Among them, HiBench suite [25] is the most famous one which is designed for testing the performance difference between Hadoop and Spark. In this section, we first present the ideas of HiBench suite, and then discuss its limitations and explain some workloads includes in its package.

Benchmarking is a set of experimental approaches that measure the effectiveness of different computer systems. HiBench suite is a set of shell scripts developed by Intel and published under Apache Licence 2 [25]. Currently HiBench suite contains 13 workloads, under four categories: Micro Benchmarks, Web Search, SQL and Machine Learning [26]. The details of all workloads can be seen from the Table 2.1.

Category	Worklodas
Micro Benchmarks	Sort Sleep TeraSort WordCount Enhanced DFSIO
Web Search	Nutch Indexing Page Rank
SQL	Scan Join Aggregation
Machine Learning	Bayesian Classification K-Means

Table 2.1: Benchmark workloads modified from [27]

2.3.1 Micro Benchmarks

As the most popular examples, Micro Benchmarks are applied widely in the Hadoop community and some of them even being included in the Hadoop example package, and therefore becomes the part of HiBench suite. Micro Benchmarks workloads represent the perfect examples of the real-world MapReduce application [28]. One programme transforming the data from one format to another, and another withdrawing the interesting data from big volume of data.

2.3.2 Web search

The Nutch Indexing workload [29] is an indexing subsystem from Nutch, one popular open-source searching engine. As one of the most significant use of MapReudce, Nutch indexing uses the web data whose words and hyperlinks follows the normal distribution with related parameters. The program to produce the texts of web page is the default Linux dictionary files.

The Page Rank workload is a test case from SmartForg, an open-source systems management framework. It is an algorithm implemented based on *Spark-Mllib*[1] which is widely used to rank the web pages based on the numbers of the reference links [7]. Also, the hyperlinks and words follow the normal distribution.

[1] A package to implement machine learning algorithm for Spark

2.3.3 SQL

This workload is implemented under hive and is for testing the effectiveness of SQL queries.

Scan query: the purpose of the Scan query is to select and project the relational tables.

Aggregation Query: The purpose of the aggregation query is to firstly group the relational tables, parse each tuple and finally perform a high-cardinal aggregation operation.

Join Query: The purpose of Join query is to join the dimension table and then sort the results.

2.3.4 Machine learning

The implementation of the *MLib* algorithm is in *Mahout*[2], an open-source *MLib* library implemented on the Hadoop cluster. As another important use of Hadoop, these algorithms are included in HiBench suite.

Naïve Bayes is a classification algorithm with multiple class which can provides each pair of features independent assumption. This workload is implemented based on *Spark.Mill ib* and use the text file generated automatically. Also, the words of the text files follow the *Gussian distribution.*

K-mean is a famous algorithm for data mining and knowledge discovery. The input data are samples which represented by a vector with a numerical dimension [25]. This algorithm is implemented based on *Spark.Millib* and its input data follows uniform and *Guassian distribution*[3].

2.3.5 Benchmarking Methodology

Figure 2.3 briefly explains the different steps in HiBench Suite Methodology. In the initial stage, all the software components need to be installed and configured properly (*Java, Mahout, CDH, HiBench, DSE* [30]). Then in stage 2, which is called workload preparation, parameters related to workloads need to be defined and the test data need to be created. The parameters as well as generated data will be utilized as the input of the stage 3(workload execution). Each workload will be executed three times to make sure the representativeness of the results, which means the data generated in stage 2 and the Workload Execution in stage 3 will be repeated three times. Before conducting another new experiment in workload preparation stage, the existing data will be removed and the new one will be created. After stage 3 finished, HiBench will provide a report to the user which includes two important information: *Duration* and *Throughput.* The *Duration* is calculated by the end time to reduce the start time while the *Throughput* is calculated by dividing the sizes of the input data during the *Duration* time. Based on the report, the user can analysis the results and present them in different ways.

[2]The machine learning algorithm library for Haddop MapReduce.
[3]Another name of normal distribution.

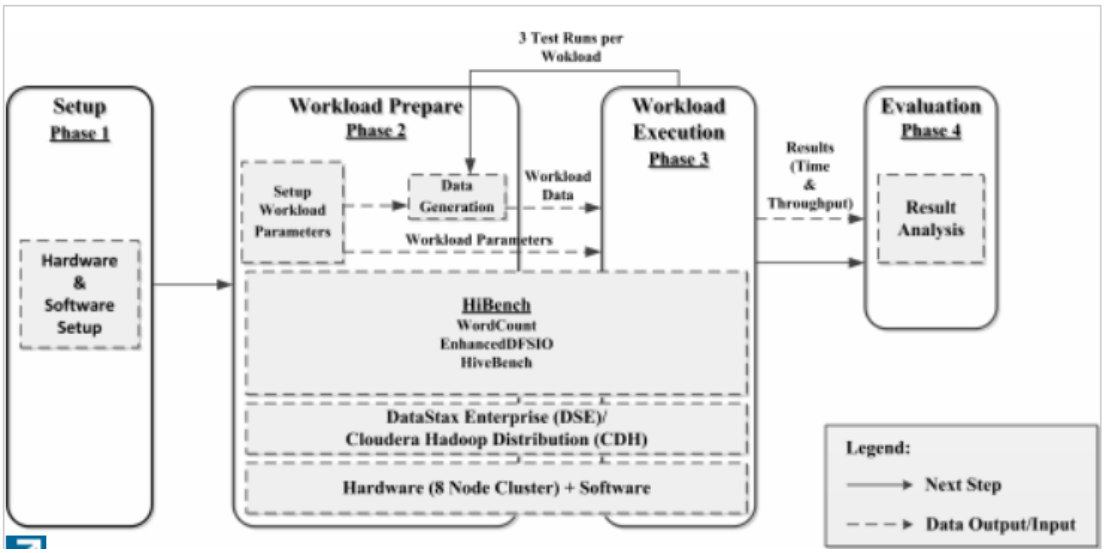

Figure 2.3: Benchmarking methodology process diagram from [30]

Lisiting 2.1 illustrates the configuration of Hadoop from the first stage. From the listing, *hibench.hadoop.home* and *hibench.hdfs.master* needs to be set that represent the directory of Hadoop and the address of HDFS respectively. Also, the Hadoop version needs to be set by *hibench.hadoop.release*.

```
# Hadoop home
hibench.hadoop.home      /usr/hdp/2.6.0.3-8/hadooop

# The path of hadoop executable
hibench.hadoop.executable      ${hibench.hadoop.home}/bin/hadoop

# Hadoop configraution directory
hibench.hadoop.configure.dir  ${hibench.hadoop.home}/etc/hadoop

# The root HDFS path to store HiBench data
hibench.hdfs.master       hdfs://it066431:8020://user/yjliu/


# Hadoop release provider. Supported value: apache, cdh5, hdp
hibench.hadoop.release    hdp
```

Listing 2.1: Hadoop configuration in stage1 modified from [27]

Listing 2.2 presents the configuration of Spark from the first stage. Among all the settings, *hibench.spark.home* is the most important setting which points to the position of Spark. Also, there are several options for the users to set to fit the sizes of their jobs that are: "*hibench.yarn.executor.num* , *hibench.yarn.executor.cores* ,*spark.executor.memory* and *spark.driver.memory*" [27].

```
# Spark home
hibench.spark.home      /usr/hdp/current/spark2-client

# Spark master
#   standalone mode: spark://xxx:7077
#   YARN mode: yarn-client
hibench.spark.master    yarn-client

```

```
# executor number and cores when running on Yarn
hibench.yarn.executor.num     2
hibench.yarn.executor.cores   4

# executor and driver memory in standalone & YARN mode
spark.executor.memory  4g
spark.driver.memory    4g

# set spark parallelism property according to hibench parallelism value
spark.default.parallelism     ${hibench.default.map.parallelism}

# set spark sql default shuffle partitions according to hibench parallelism value
spark.sql.shuffle.partitions  ${hibench.default.shuffle.parallelism}
```

Listing 2.2: HiBench configuration in stage2 modified from [27]

Listing 2.3 presents some other configuration for HiBench in the second stage. Among them, *hibench.scale.profile* is the most important which controls the sizes of the workloads. Also, *hibench.default.map.parallelism* and *hibench.default.shuffle.parallelism* are two important parameters which represent the number of map tasks and reduce tasks for the workloads. Besides, the *report files* section provides some information that the users need after the jobs finishes.

```

# Data scale profile. Available value is tiny, small, large, huge, gigantic and bigdata.
# The definition of these profiles can be found in the workload conf file i.e. conf/
    workloads/micro/wordcount.conf
hibench.scale.profile                 tiny
# Mapper number in hadoop, partition number in Spark
hibench.default.map.parallelism         8

# Reducer nubmer in hadoop, shuffle partition number in Spark
hibench.default.shuffle.parallelism     8


#========================================================
# Report files
#========================================================
# default report formats
hibench.report.formats          %-12s %-10s %-8s %-20s %-20s %-20s %-20s\n

# default report dir path
hibench.report.dir              ${hibench.home}/report

# default report file name
hibench.report.name             hibench.report

# input/output format settings. Available formats: Text, Sequence.
sparkbench.inputformat          Sequence
sparkbench.outputformat         Sequence

# hibench config folder
hibench.configure.dir           ${hibench.home}/conf

# default hibench HDFS root
hibench.hdfs.data.dir           ${hibench.hdfs.master}/HiBench

# path of hibench jars
hibench.hibench.datatool.dir            ${hibench.home}/autogen/target/autogen-7.1-SNAPSHOT-
    jar-with-dependencies.jar
hibench.common.jar                      ${hibench.home}/common/target/hibench-common-7.1-
    SNAPSHOT-jar-with-dependencies.jar
hibench.sparkbench.jar                  ${hibench.home}/sparkbench/assembly/target/
    sparkbench-assembly-7.1-SNAPSHOT-dist.jar
hibench.streambench.stormbench.jar      ${hibench.home}/stormbench/streaming/target/
    stormbench-streaming-7.1-SNAPSHOT.jar
```

```
39 hibench.streambench.gearpump.jar          ${hibench.home}/gearpumpbench/streaming/target/
       gearpumpbench-streaming-7.1-SNAPSHOT-jar-with-dependencies.jar
40 hibench.streambench.flinkbench.jar        ${hibench.home}/flinkbench/streaming/target/
       flinkbench-streaming-7.1-SNAPSHOT-jar-with-dependencies.jar
```

Listing 2.3: Spark configuration in stage1 modified from [27]

Listing 2.4 presents the Wordcount configuration in stage 2. From the listing, we can control or create any sizes of datasets we want. Also, we can provides the input and output directories to extend the experiments.

```
1  #datagen
2  hibench.wordcount.tiny.datasize                 32000
3  hibench.wordcount.small.datasize                320000000
4  hibench.wordcount.large.datasize                3200000000
5  hibench.wordcount.huge.datasize                 32000000000
6  hibench.wordcount.gigantic.datasize             320000000000
7  hibench.wordcount.bigdata.datasize              1600000000000
8
9  hibench.workload.datasize                       ${hibench.wordcount.${hibench.scale.profile}.
       datasize}
10
11 # export for shell script
12 hibench.workload.input                          ${hibench.hdfs.data.dir}/Wordcount/Input
13 hibench.workload.output                         ${hibench.hdfs.data.dir}/Wordcount/Output
```

Listing 2.4: WordCount configuration in stage2 modified from [27]

Listing 2.5 and 2.6 presents the process of data generation and data processing. From the figure, we can find the MapReduce command as well as the operations on the HDFS. In addition, listing 2.7 illustrates the format of the report after finished several workloads.

```
1  patching args=
2  Parsing conf: /home/yjliu/Hibench/HiBench-master/conf/hadoop.conf
3  Parsing conf: /home/yjliu/Hibench/HiBench-master/conf/hibench.conf
4  Parsing conf: /home/yjliu/Hibench/HiBench-master/conf/spark.conf
5  Parsing conf: /home/yjliu/Hibench/HiBench-master/conf/workloads/micro/wordcount.conf
6  probe sleep jar: /usr/hdp/current/hadoop-client/../hadoop-mapreduce/hadoop-mapreduce-client-
       jobclient-tests.jar
7  start HadoopPrepareWordcount bench
8  hdfs rm -r: /usr/hdp/current/hadoop-client/bin/hadoop --config /usr/hdp/current/hadoop-
       client/etc/hadoop fs -rm -r -skipTrash hdfs://it066427:8020//user/yjliu//HiBench/
       Wordcount/Input
9  Deleted hdfs://it066427:8020/user/yjliu/HiBench/Wordcount/Input
10 Submit MapReduce Job: /usr/hdp/current/hadoop-client/bin/hadoop --config /usr/hdp/current/
       hadoop-client/etc/hadoop jar /usr/hdp/current/hadoop-client/../hadoop-mapreduce/hadoop-
       mapreduce-examples.jar randomtextwriter -D mapreduce.randomtextwriter.totalbytes=32000 -
       D mapreduce.randomtextwriter.bytespermap=3200 -D mapreduce.job.maps=10 -D mapreduce.job.
       reduces=20 hdfs://it066427:8020//user/yjliu//HiBench/Wordcount/Input
11  The job took 12 seconds.
12 finish HadoopPrepareWordcount bench
```

Listing 2.5: Data preparation in stage2 modified from [27]

```
1  yjliu@it066427:~/Hibench/HiBench-master/bin/workloads/micro/wordcount/hadoop$ ./run.sh
2  patching args=
3  Parsing conf: /home/yjliu/Hibench/HiBench-master/conf/hadoop.conf
4  Parsing conf: /home/yjliu/Hibench/HiBench-master/conf/hibench.conf
5  Parsing conf: /home/yjliu/Hibench/HiBench-master/conf/spark.conf
6  Parsing conf: /home/yjliu/Hibench/HiBench-master/conf/workloads/micro/wordcount.conf
7  probe sleep jar: /usr/hdp/current/hadoop-client/../hadoop-mapreduce/hadoop-mapreduce-client-
       jobclient-tests.jar
```

```
8  start HadoopWordcount bench
9  hdfs rm -r: /usr/hdp/current/hadoop-client/bin/hadoop --config /usr/hdp/current/hadoop-
       client/etc/hadoop fs -rm -r -skipTrash hdfs://it066427:8020//user/yjliu//HiBench/
       Wordcount/Output
10 Deleted hdfs://it066427:8020/user/yjliu/HiBench/Wordcount/Output
11 hdfs du -s: /usr/hdp/current/hadoop-client/bin/hadoop --config /usr/hdp/current/hadoop-
       client/etc/hadoop fs -du -s hdfs://it066427:8020//user/yjliu//HiBench/Wordcount/Input
12 Submit MapReduce Job: /usr/hdp/current/hadoop-client/bin/hadoop --config /usr/hdp/current/
       hadoop-client/etc/hadoop jar /usr/hdp/current/hadoop-client/../hadoop-mapreduce/hadoop-
       mapreduce-examples.jar wordcount -D mapreduce.job.maps=10 -D mapreduce.job.reduces=20 -D
        mapreduce.inputformat.class=org.apache.hadoop.mapreduce.lib.input.
       SequenceFileInputFormat -D mapreduce.outputformat.class=org.apache.hadoop.mapreduce.lib.
       output.SequenceFileOutputFormat -D mapreduce.job.inputformat.class=org.apache.hadoop.
       mapreduce.lib.input.SequenceFileInputFormat -D mapreduce.job.outputformat.class=org.
       apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat hdfs://it066427:8020//user/
       yjliu//HiBench/Wordcount/Input hdfs://it066427:8020//user/yjliu//HiBench/Wordcount/
       Output
13 ECDSA key fingerprint is SHA256:4+Fjz1zLl8kUMiJfdCnpL3CUqHY+OOBj7fQmQ3akvUQ.
14           Bytes Written=23016
15 finish HadoopWordcount bench
```

Listing 2.6: Wordcount in Hadoop in stage 3 modified from [27]

```
1  Type Date Time Input_data_size Duration(s) Throughput(bytes/s) Throughput/node
2  ScalaSparkKmeans 2018-10-16 18:31:29 1396221 20.048 69643 6964
3  ScalaSparkTerasort 2018-10-16 19:00:10 3200000 20.985 152489 15248
4  ScalaSparkKmeans 2018-10-16 19:04:47 1396221 19.894 70183 7018
5  ScalaSparkKmeans 2018-10-16 20:47:08 1396221 19.206 72697 7269
6  ScalaSparkKmeans 2018-10-17 12:11:36 1396221 16.959 82329 8232
7  ScalaSparkWordcount 2018-11-02 16:36:22 37270 17.751 2099 209
8  HadoopTerasort 2018-11-02 17:36:07 3200000 30.919 103496 10349
9  ScalaSparkTerasort 2018-11-02 17:40:11 3200000 21.515 148733 14873
10 ScalaSparkTerasort 2018-11-02 20:43:59 3200000 21.027 152185 15218
11 ScalaSparkTerasort 2018-11-05 13:08:13 3200000 20.359 157178 15717
12 ScalaSparkWordcount 2018-11-05 15:35:55 36232 15.329 2363 236
13 ScalaSparkTerasort 2018-11-05 17:07:44 3200000 21.339 149960 14996
14 ScalaSparkTerasort 2018-11-05 18:15:05 3200000 19.830 161371 16137
15 ScalaSparkWordcount 2018-11-06 16:04:35 36232 37.314 971 97
```

Listing 2.7: Restuls in stage 3 modified from [27]

2.3.6 Performance metrics

HiBench results forms are under these essential metrics [31] which are:

- Input_data_size: It is the size of the data generated in stage 2 [32]. HiBench provides several datasets for the users to test the clusters which are *tiny, small, large, huge, gigantic* and *bigdata*. Each of the dataset represent one kind of data volume and the users can choose the proper one based on the scale of the cluster.

- Duration: it describes a period of time that the workload is operating. It is just the difference between start time and endtime of the workloads. It is calculated by seconds.

- Throughput: there are two kinds of *Throughput* provided by HiBench, one is *Throughput* for the whole cluster and one is the *Throughput* for different nodes. These pa-

rameters present how much information the cluster can process in a given period of time [31]. They are measured by *bytes/second*.

2.4 Discussion

Since Bechmarking tests can be seen as the foundation of the quantitative computer system research, these workloads should be distinctive to represent the characteristics of the target system as well as be various enough to present the behaviour scope of the target application [25]. Although, HiBench suite program as described above shows its diversity and representativeness through its examples, there are still some limitations that leads it to fail to present the cluster performance from many the aspects like control the sizes of workloads or monitor the resource utilization of the cluster.

In particular, though HiBench suite attempts to include all sizes of the cluster in its experiment, they do not provide enough datasets for different users to test. Until now, there are only six dataset options provided by HiBench: *tiny, small, large, huge,gigantic* and *bigdata* and the sizes of the datasets ranges from 30 kb to 1.5 Tb. Compared to other tests, the scalability can be poor because all of these values are fixed in the experiments. Also, the generated data is under sequencefile[4] format which is invisible to the users. The users need to convert them to text files or use some specific software to check the content of the generated data. This setting brings much inconvenience to the users in their daily operation. Figure 2.4 shows the K-means data under sequencefile format and listing 2.8 shows that opened by Mahout.

Figure 2.4: Generated K-means data under sequencefile format

```
1 yjliu@it066427:~/mahout-distribution-0.8$ bin/mahout seqdumper -i /user/yjliu/HiBench/
      Kmeans2/Input/cluster/part-00000
2 Running on hadoop, using /usr/bin/hadoop and HADOOP_CONF_DIR=
3 MAHOUT-JOB: /home/yjliu/mahout-distribution-0.8/mahout-examples-0.8-job.jar
```

[4]A kind of flat file under key-value pair format.

```
4  19/02/21 15:21:11 INFO common.AbstractJob: Command line arguments: {--endPhase=[2147483647],
       --input=[/user/yjliu/HiBench/Kmeans2/Input/cluster/part-00000], --startPhase=[0], --
       tempDir=[temp]}
5  Input Path: /user/yjliu/HiBench/Kmeans2/Input/cluster/part-00000
6  Key class: class org.apache.hadoop.io.Text Value Class: class org.apache.mahout.clustering.
       kmeans.Kluster
7  Key: CL-0: Value: CL-0{n=0 c=[991.089, 311.404, 479.975] r=[]}
8  Key: CL-1: Value: CL-1{n=0 c=[323.392, 230.470, 948.572] r=[]}
9  Key: CL-2: Value: CL-2{n=0 c=[93.758, 947.342, 828.972] r=[]}
10 Key: CL-3: Value: CL-3{n=0 c=[182.601, 717.132, 629.741] r=[]}
11 Key: CL-4: Value: CL-4{n=0 c=[737.270, 576.215, 476.047] r=[]}
12 Count: 5
13 19/02/21 15:21:12 INFO driver.MahoutDriver: Program took 1385 ms (Minutes:
       0.023083333333333334)
```

Listing 2.8: Generated K-means data opened by Mahout

In addition, the workloads contained in both Spark and MapReduce benchmark can only provide several parameters for users to test: 2 for MapReduce and 6 for Spark. In fact, the parameters for Spark and MapReduce can improve the efficiency of the jobs dramatically in most of the circumstances. Thus, the accuracy of the experiments needs to be discussed further.

On the other hand, the HiBench result form only includes 8 items while only four of them present the performance of the cluster. Also, for MapReudce benchmarks, HiBench does not allows Hadoop to run the jobs on Yarn which means we are unable to pick any result parameters from other platform. Although the *throughput* and *duration* are the most important aspects for the cluster, many other aspects should take into consideration to present an overall view of the jobs.

Thus, we decided to extend several workloads from the HiBench to our own cluster. Based on our research, we select Wordcount, Terasort and K-means workloads as our targets because they are the representatives of specific workloads. Wordcount stands for the aggregation job, Terasort stands for the shuffle job and K-means stands for the iterative jobs. Also, another reason is all of the workloads are already implemented under Hadoop and Spark example package. It can make sure that our experiments are not affected by the coding ability. Next, we are going to talk about the ideas of each workload and explain why there is a difference between MapReduce and Spark when apply the same workloads.

2.5 Wordcount

Wordcount is a program which counts the occurrence of each word from a text or sequencefile [30]. As one of the most classic examples, it is widely used to evaluate the aggregation performance for both MapReduce and Spark. The input data is produced by *RandomTextWriter* [33] which is a program generating large context files for users to test. It is a powerful and easy to use example not only because it is included in Hadoop example package, but also it is able to create large input files in HDFS. Also, there are

several parameters to improve the efficiency which will be mentioned later.

2.5.1 Data generation

Firstly, several parameters needs to be set before executing the examples. There are five parameters provided by *RandomTextWriter* to control the format of the input files. The first two are *minwordskey* and *maxwordskey*. These two parameters are used for setting the length of key. The next parameters are *minwordsvalue* and *maxwordsvalue* that are used to control the length of the value (The output of the mapper is sequencefile which includes key and value). The last parameter is *totalbytes* which represents the sizes of the data the user wants to generate. After setting the parameters properly, we are going to produce the input data. Listing 2.9 presents the examples of the input data.

```
1  yjliu@it066427:~/mahout-distribution-0.8$ bin/mahout seqdumper -i /user/yjliu/HiBench/
       Wordcount/Input/part-m-00000
2  Running on hadoop, using /usr/bin/hadoop and HADOOP_CONF_DIR=
3  MAHOUT-JOB: /home/yjliu/mahout-distribution-0.8/mahout-examples-0.8-job.jar
4  19/02/21 15:33:40 INFO common.AbstractJob: Command line arguments: {--endPhase=[2147483647],
       --input=[/user/yjliu/HiBench/Wordcount/Input/part-m-00000], --startPhase=[0], --tempDir
       =[temp]}
5  Input Path: /user/yjliu/HiBench/Wordcount/Input/part-m-00000
6  Key class: class org.apache.hadoop.io.Text Value Class: class org.apache.hadoop.io.Text
7  Key: acocotl overcrown mericarp uncompromisingly drome bromate stachyuraceous Hysterocarpus
       allotropic : Value: sequacity reciprocation unchatteled spermaphyte sedentariness anta
       homotransplant uncompromisingly ethnocracy tomorrowness lyrebird bladderwort flatman
       trip decardinalizesymbiogenetically absvolt atlantite subfoliar ribaldrous unscourged
       oratorize bromic devilwise friarhood abthainry coracomandibular epidymides subangulated
       migrainoid reappreciate ununiformly groundneedle pentosuria supermarket subirrigate
       frenal blightbird phytonic
8  Key: commandingness times barkometer liquidity coracomandibular abstractionism seeingness
       authorling trip : Value: phallaceous ambitus figureheadship unrepealably lebensraum
       rehabilitative various posterishness kenno metaphrastical astucious omniscribent
       approbation tonsure mustafina subofficer Triconodonta cubby culm seraphism Shiah slait
       perculsive barkometer uninhabitedness putative archesporial insatiately Animalivora
       quintette rizzomed astucious sviatonosite orgiastic Joachimite prescriber eristically
       Macraucheniidae warlike rainproof plerome dialoguer equiconvex oinomancy redescend quad
9  Key: circular unchatteled ascitic stormy danseuse tonsure Dunlop : Value: mastication
       overcrown lithotresis swoony almud hemimelus lithotresis archesporial dehairer
       venialness paranephros calycular trailmaking impressor sesquiquintile unswanlike
       iniquitously atlantite zenick mangonism diplomatize Effie valvula raphis trip flatman
       seeingness parmelioid isopelletierin ribaldrous quadrennial licitness bugre brutism
       trabecular octogynous sapphiric laurinoxylon bugre enhedge mendacity mammonish Helvidian
        shellworker meloplasty acocotl steprelationship groundneedle foursquare Jerusalem
       Pishquow erlking embryotic sloped eternal hysterolysis dunkadoo isopelletierin chooser
       taurocolla oblongly noreast pentosuria ten abstractionism mammonish Bermudian
       lophotrichic pope Passiflorales soorkee ribaldrous floatability mechanist
       uninhabitedness slait counteractively
10 Key: Hydrangea choralcelo psychofugal sialadenitis tum : Value: oflete posterishness
       cobeliever pseudohalogen unschematized Mormyrus Munychian diopside boor overcontribute
       guitarist seeingness widdle Orbitolina electrotechnics craglike iniquitously equiconvex
       outwealth arrowworm Lentibulariaceae dastardliness elemicin ticktick karyological naught
        biventer Bushongo lineamental groundneedle arval commotion oinomancy redesertion
       Bulanda entame pope oversand pamphlet vesperal flutist pomiferous topline sequestrum
       sportswomanship unpeople Ophiosaurus slangy returnability ordinant critically
       psychofugal monogoneutic basto subtransverse comism antalgol bot rebilling Pishquow
       penult Glecoma squit scyphostoma silicize craglike decidable appetible reciprocation
       bromic predebit comparability stiffish comism inventurous peptonate elemicin
       orchiocatabasis unefficient idiotize pamphlet Alethea naught Aktistetae bromic
       homotransplant paradisean sheepskin perculsive superindifference Confervales
       unreprimanded Isokontae dishpan raphis nonlustrous flutist culm
11 Key: bespin macropterous bismuthiferous prolificy frictionlessly coldfinch adscendent
       inferent tautness : Value: digitule Alethea chalcites skyshine wingable manilla
       alveolite arsenide seelful oflete tonsure emir unimmortal subtransverse unexplicit
       yawler biopsic unexplicit stereotypography overcrown precostal
12 Count: 5
```

```
13 19/02/21 15:33:41 INFO driver.MahoutDriver: Program took 1267 ms (Minutes:
       0.021116666666666666)
14 yjliu@it066427:~/mahout-distribution-0.8$
```

Listing 2.9: The input data for Wordcount

2.5.2 Algorithm

Next, we are going to talk about the parallel algorithm used on the Wordcount example. When concerning the parallel algorithm, divide and rule strategy will come to our mind which means assign the operation on the large-scale datasets to each work node under the control of a master node, and then integrates the intermediate results of each node to obtain the final result. In other words, it is about breaking the tasks and aggregate the results. Wordcount adopts this strategy and works like that: the input files in HDFS will be divided into different splits and transfer to different map tasks based on their Hash value. Then the map tasks will sort them and write the data into different partitions. The reducers will receive the data from different partitions and sort them as the final results. Finally, the results will be output to the HDFS. Figure 2.5 [34] illustrates how the parallel algorithm works.

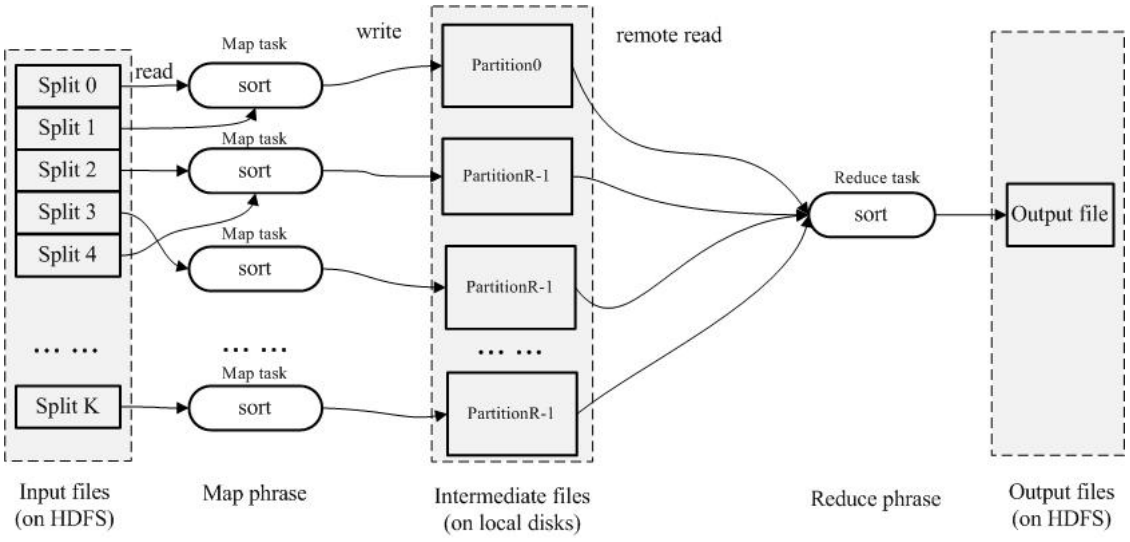

Figure 2.5: Parallel algorithm for Wordcount from [34]

2.5.3 Wordcount in MapReduce

Then we are going to explain the examples implemented by Hadoop MapReduce. The source code is from MapReduce examples package. After we generated the input data and store it on the HDFS, we are going to implement the example.

Inputsplit stage: before mapper tasks begin, MapReduce calculates the inputsplits according to the sizes of input files. The inputsplits is not for storing the data, but an

array of slice lengths and the location where the data is recorded. Each inputsplit match one mapper tasks. Thus, the number of Mapper tasks and when Mapper tasks begin and end decided in this process. Also, the input files are divided into different splits and transfer to different Mappers.

Map stage: MapReduce reads the splits line by line and turn each line as a key-value pair. The key is offset which represents the position of the line while the value is the content of the line. Next, Mappers will assign the key value pairs to different map tasks which means the map function begins. Each map tasks divide the values by space and count the new key value pairs of each elements as one. Finally, Mapper will sort these key value pairs based on their keys. Figure 2.6 [1] presents the whole process.

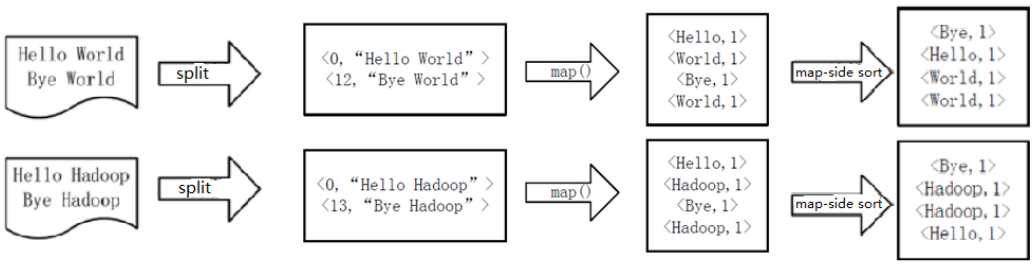

Figure 2.6: Splits and Map stage modify from [1]

Combiner stage: combiner stage is optional and is similar to reduce stage. Actually, it is a local reduce operation which accumulate the same value of the key to get the final output of the Mapper. Figure 2.7 [1] shows the combine process.

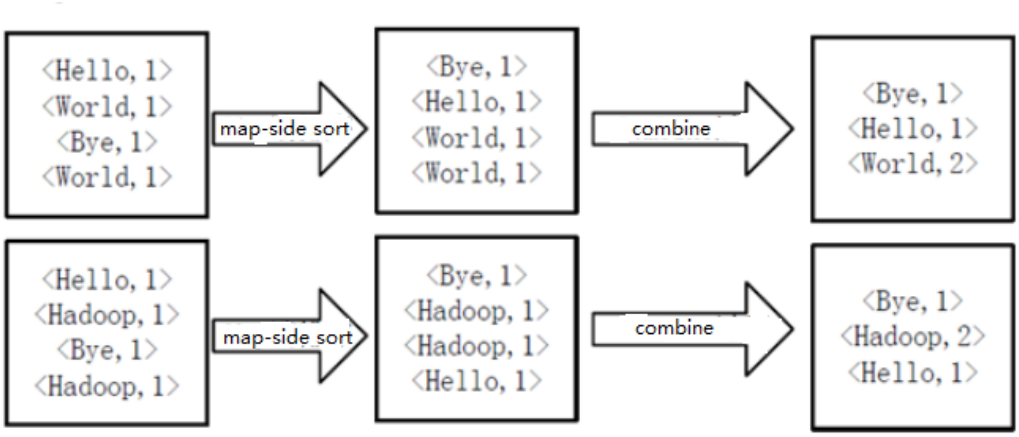

Figure 2.7: Combiner stage modify from [1]

Reduce stage: reducer accepts the data through shuffle and sort the data. It is called reducer-side sort which means aggregate the key-value pairs with the same key and gener-

ate new key-value pairs. The new key value pairs use the words as its key while building a list which contains the occurrence of the element from different map tasks. Then the reducer assigns the data to different reduce tasks. Each of the reduce task will calculate the occurrence of each words by accumulating each element from its value and generate final key value pairs to save the result. Finally, the result will be written back to HDFS. Figure 2.8 [1] illustrates the reduce stage.

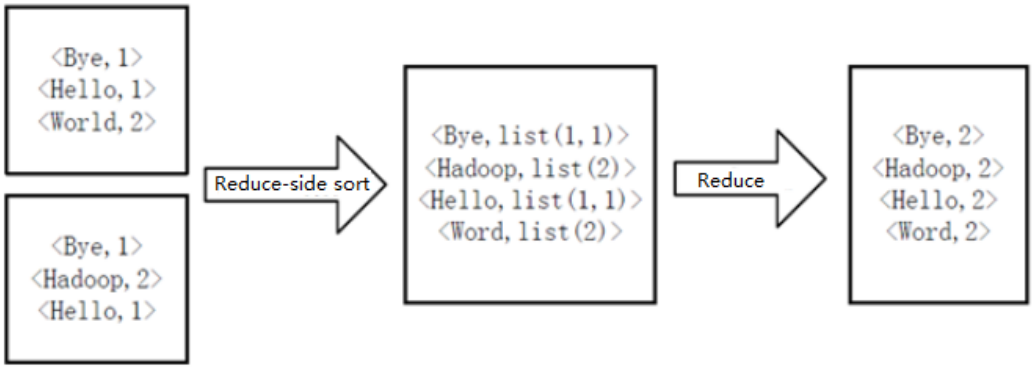

Figure 2.8: Reduce stage modify from [1]

2.5.4 Wordcount in Spark

Compared to MapReduce framwork, Spark adopts a new form of dataset called Resilient Distributed Datasets (RDDs) [16] which allows the users to save the data into memory. Thus, all the Spark operation is based on memory or RDDs and we are going to discuss the WordCount program provided by Spark examples in details. Figure 2.9 [35] presents how Spark works on WordCount.

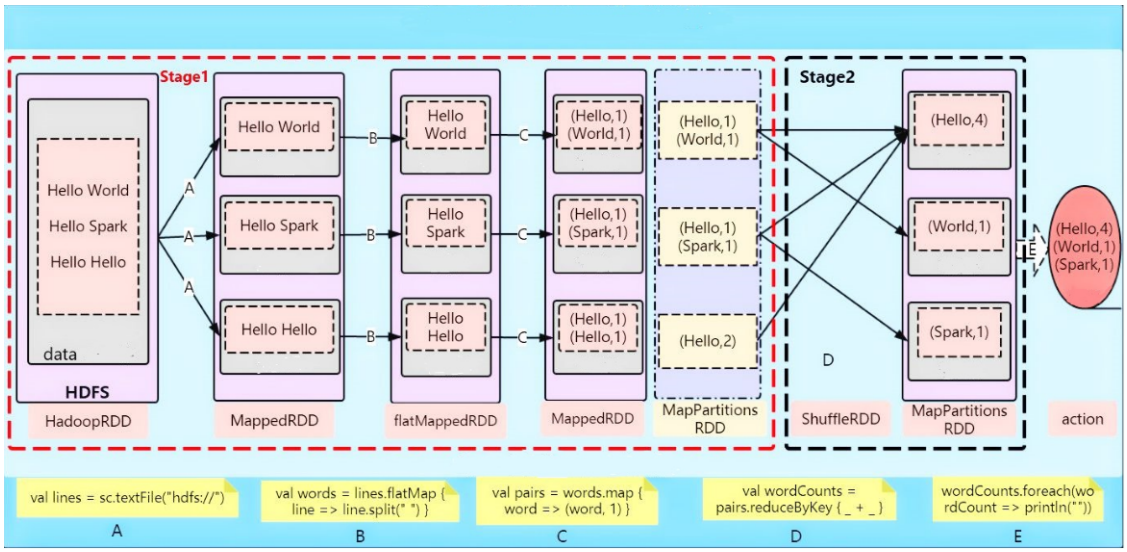

Figure 2.9: How Spark works from [35]

From the figure, we can get that there are two stages exists when we implement the WordCount example. As we explain before, there are two kinds of operations in Spark: *transformation* and *action*. The stages will be divided once the *action* operation happens.

For stage 1, the first step is loading the data into Spark and turn it into RDDs. From the figure, the example uses *textFile* operator to read the files from HDFS. The input files will be read by line and each of the line will become one element of the RDD. Finally, RDD is converted from HDFS files to MappedRDD and its data type is string.

The next step applies *flatmap* operator aimed at processing each element in MappedRDD. The *flatmap* belongs to *transformation* operation which allows turning the *map* datasets to *flatmap* datasets. In this example, each line of the MappedRDD is split by a space to get an array of words, and then the array is flattened to form a string. Finally, all of the strings are saved in the FlatMappedRDD.

Then, we use *map* operator to mark all the elements in FlatMappedRDD which means reformat each element from string to key value pair.There,RDD is converted from FlatMappedRDD to MapparatitionRDD.

The following step uses *reducebykey* operator to merge the value of each pair elements with the same key from different RDDs. This operation will drag the pairs with same key together and accumulate the value to get the occurrence of each element. RDD there is converted from MapparatitionRDD to shuffleRDD and then turn to MappedRDD.

The last step uses *foreach* operator which can through all the elements in MappedRDD and print the data on the screen. Since *foreach* operator belongs to action operation, all the jobs will be triggered after running this kind of operation. Also, we can implement other operation like *count*, *saveastextfile* to meet our requirements.

2.6 TeraSort

1TB sorting [36] is used to measure the data processing capabilities of distributed computing frameworks. As one of the Hadoop sorting jobs, TeraSort helped Hadoop win first place in 1TB sorting benchmark evaluation in 2008 and it only costs 209 s. Thus, Hadoop includes the source code of TeraSort in its example package in order to facilitate the users to evaluate the performance of their clusters. The input data is generated by *TeraGen*[5], a data generating function implemented in Java. In addition, Hadoop also provides the function to validate the sorting result called *TeraValidate*. Although Terasort is a small gadget, it can still provide many hints to improve the cluster performance.

2.6.1 Data generation

Before explaining how TeraSort works, we will briefly introduce something related to the input data. Firstly, the input file is composed by different rows of 100-byte records. Since each record counts 100 bytes, the users can calculate the length of their input file easily. Also, each record consists of three parts: the first ten bytes are ten binary code random characters and it will be used as key for sorting. Then, the next ten bytes are the id of the row followed by 8 segments of 10-byte identical random uppercase letters record. Besides, there are two parameters we need to set before running the program which are the length of the input file and the output address folder. Figure 2.10 presents the format of the input data.

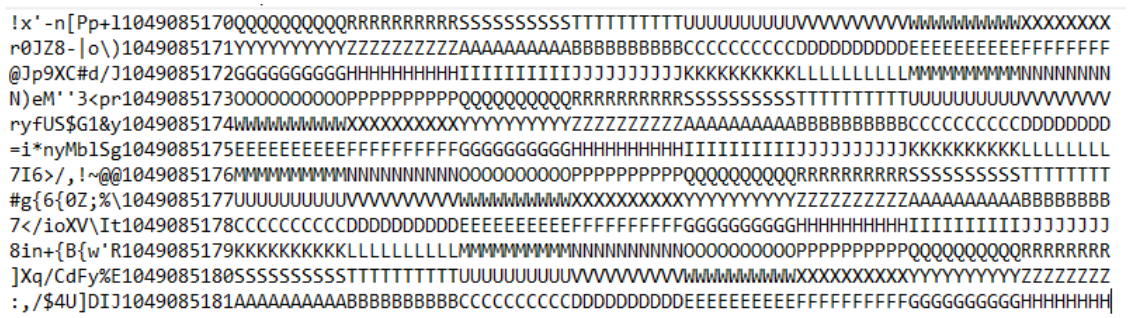

Figure 2.10: Input data for Terasort

2.6.2 Algorithm explaination

As we mentioned above, the standard parallel algorithm can guarantee a high degree of parallelism in the map stage, but no parallelism in the reduce stage. In order to improve the parallelism in the reduce stage, TeraSort utilizes the way to improves algorithm : during the map stage, each map task will split the input data into R partitions which is same to the number of reduce tasks. Then it needs to make sure that all the data in

[5]The function provided by Hadoop to create the input data.

partition I is larger than that in paratition I+1. After that, during the reduce stage, the reduce tasks I get the data from partition I and sort the data in partition I. Thus, the results in reduce task I is larger than that in reduce task I+1. Finally, it outputs the reduce results and we can get the final outcome. Figure 2.11 shows how the TeraSort algorithm works.

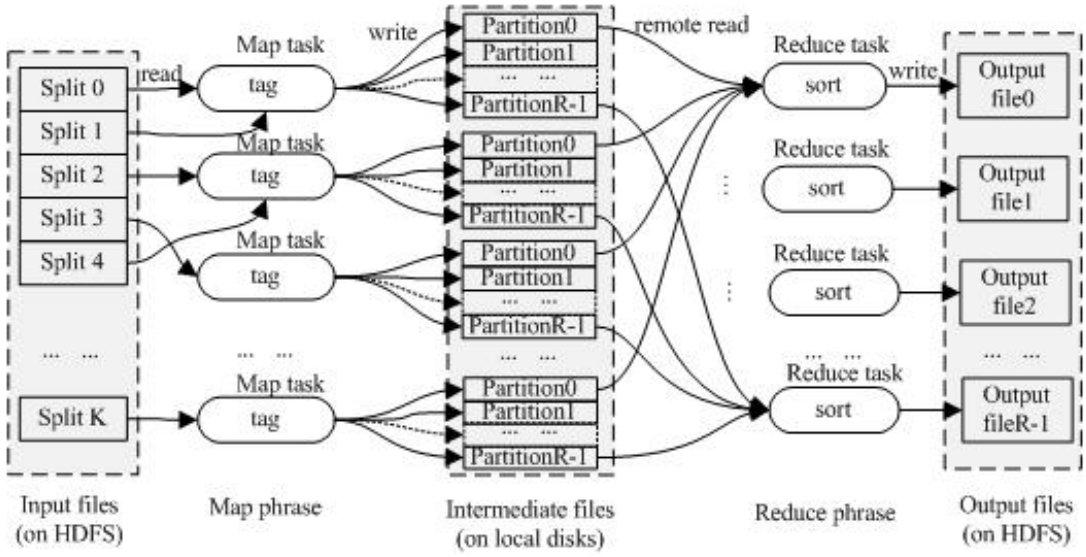

Figure 2.11: Algorithm for Terasort from [34]

Although this algorithm shows high parallelism in reduce stage, it increases the shuffle pressure for the cluster because of too many partitions created in map stage. Also, there are two technical difficulties to implement the algorithm: one is how to determine the range of R partition for each map task. The other is how to quickly determine which partition it belongs to for one specific record.

2.6.3 TeraSort in MapReduce

As we all know, the main feature of MapReduce is sorting. It happens in both map and reduce stages. In other words, all the data in different partitions are in order. Thus, TeraSort only needs to do one thing: ensure the partitions are in order. To implement the algorithm above, Hadoop improves its partition strategy and adopts *totalorderpartition*[6]. *TeraSort* modifies the *totalorderpartition* and implements its own one in three steps: the first step is sampling, the second step is marking the records during the map stage and finally conduct sorting during the reduce stage [37].

For the first stage, data sampling is performed on the *JobClient*. At first, a part of the data is extracted from the input file and the frequency is determined by *tera-*

[6]The partition strategy implemented by Hadoop

sort.partitions.sample for which the default value is 100,000. After sampling enough records, MapReduce will sort these data and divide it into R partitions to find the upper and lower lines of each partition (called the "split point"). Then these split points will be saved on the distributed cache named as _partition.lst.

During the map stage, each map task reads the split points from the distributed cache, and establishes a trie tree [38][7] (two layers of trie trees, the leaf node of the tree stores the corresponding reduce task number). After that, MapReduce starts processing the data. For each record in different partitions, find the number of the reduce task it belongs to in the trie tree and save it in this partition.

Next, in the reduce stage, each reduce task reads its corresponding data from the partitions and conduct local sorting. Finally, the result of the reduce task is sequentially output according to the reduce task number.

Here is an example to explain how to use trie trees to assign records to different reduce tasks.

Imagine the number of sample data is 100,000 and the numbers of splits is 4. Thus, we need to pick 25,000 samples from each split. Then if the sample data is like this: "b, abc, abd, bcd, abcd, efg, hii, afd, rrr, mnk" [34]. After sorting the data, we get: "abc, abcd, abd, afd, b, bcd, efg, hii, mnk, rrr" [34]. If the reduce tasks is four, we can get the split points: abd,bcd, mnk. These splits points will be saved on distributed memory and now we are building the trie tree. The trie tree is shown in Figure 2.12.

From the trie tree, we can get that all the sample data is sorted and each partition is in order. In other words, all the data in split 1 is larger than that in 2, 3 and 4. Then we just need to build the trie tree on the different map tasks to assign each record to different partitions. Finally, the reduce tasks just need to sort the data in the partition and output the partitions in order to get the final result.

According to the source code provided by Hadoop, there are no any specific mapper and reducer functions in TeraSort job which means Hadoop will use default mapper and reducer function: *IdentityMapper* and *IdentityReducer*. These two functions utilize the sorting process during the map and reduce stages while not setting any operation for map and reduce tasks. Thus, the modified *totalorderpartition* will control the job and help Hadoop sort all the data quickly.

[7] A kind of method to find the words.

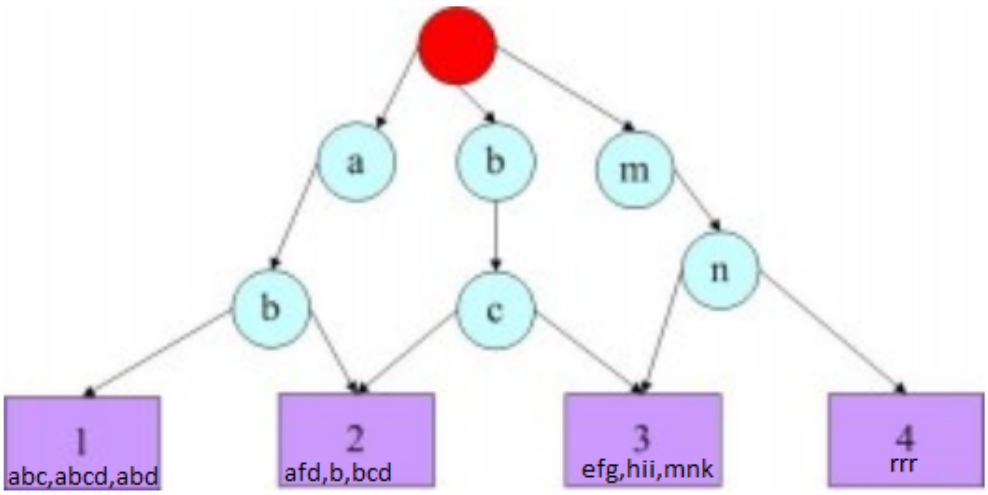

Figure 2.12: Trie tree from [34]

2.6.4 TeraSort in Spark

Compared to Mapreduce, Spark adopts two kinds of partition tools which are called *HashPartition* and *RangePartition* respectively. For *HashPartition*, the principle is like this: for a given key, Spark will calculate its hashcode and divide it by the number of partitions. The remainder is the partition ID. If the remainder is less than 0, use the remainder add the number of partitions to get the partition ID. Based on that strategy, we can find that *HashPartition* may causes uneven data volume in each partition, and in extreme cases, one partitions may own all the data in RDD. To handle the situation, *RangePartition* is proposed by Spark.

RangePartition is mainly used for sorting the data in different RDDs and the principle is like this: the first step is extracting the sample data from the entire RDD. After sort the data, calculate the maximum key of each partition and form a variable *rangeBounds* under the Array[key] format. The second step is determining the range of the key in *rangeBounds* and give the subscript id of the key in the next RDD.

RangePartition adopts *Reservoir Sampling* [39][8] which is aimed at solving the problem: select k samples from the set S containing n items where n is an infinite or unknown number. This algorithm is efficient for the case that all n items cannot be stored in main memory. The principle of the algorithm is like that: according to Dictionary of Algorithms and Data Structures, the first step is extracting the first k items from set S and putting them into the "water pond". For each S[j] term ($j \geq k$) , randomly generate an integer r ranging from 0 to j. If $r < k$, replace the item in the pond with the S[j] item.

Here is an example to explain how it works:

[8] A classic sampling method.

Imagine there are 10,000 numbers and we need to extract ten random numbers from them. We can record the sample set as S which includes 10,000 numbers. Also, we call the array to save the ten random number as R. At first, take the first ten numbers from S and fill them into the R. Then go to the first iteration. The first iteration starts from the eleventh number (subscript 10) and generate a random integer j from 0 to 10. If j $<$ 10 (j=4), we will use the 11th item in the S (S[10]) to replace the fifth item in the R (R[4]). All the iterations works like that until the end of the S.

Next, we are going to explain how to get the borders of each partition based on the Figure 2.13.

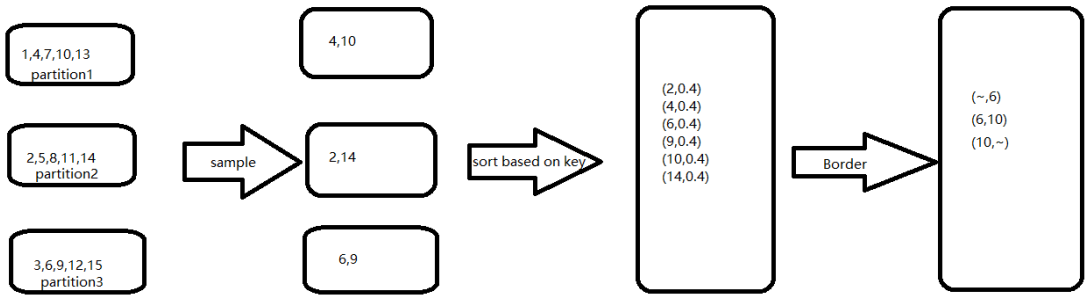

Figure 2.13: How to determine the border

From the Figure 2.13, we can see that there are three partitions and each of them owns 5 numbers. Thus, based on *Reservoir Sampling*, we pick two sample points for each partition. Then we sort them based on their key and calculate their weights which equals to the numbers of sample data divide the numbers in partition. After that, we are going to count the sum of the total weights (2.4) and use the value to divide partition number (3) to get the step (0.8). Finally, we are going to accumulate the weight for each sample data and once the value is larger than multiples of the steps we will save the key as the border. Here we get the 6 and 10 as the border which can help the each record find this partition.

Spark implements *RangePartition* in the following way:

1. Calculate the number of partitions if it is larger than one. Otherwise, return an empty array.

2. Calculate the data sample size. The rule is: at least 20 records per partition or at most 1Mb per partition.

3. Calculate *sampleSizePrePartition* based on *sampleSize* and number of partitions.

4. Call the *RangePartitioner*'s sketch function to sample the data and calculate the sample for each partition.

5. Calculate the overall proportion of the sample and the partition with too much data to avoid data skew.

6. Call the RDD sample function to re-extract the data for the partition with too much data.

7. The final sample data is sorted and distributed by *determineBounds* function to calculate *rangeBounds*.

2.7 K-means

K-means belongs to hard cluster algorithm and is aimed at " partitioning N objects into K clusters where each object belongs to the cluster with the nearest mean" [40]. This algorithm first randomly selects k objects and each of them stands for the centre or mean of the cluster. Then it will classify the other objects to different clusters based on calculating the distance between the centre and the objects [40]. Finally, it will recalculate the mean of each clusters. This process will repeat again and again until criterion function becomes convergent.

In addition, K-means is a typical distance-based clustering algorithm. The distance is used to evaluate the similarity [41]. The closer the distance between two objects, the greater the similarity will be. Besides, the algorithm reassigns each object to the nearest cluster in each iteration based on its distance from each cluster centre. When one iteration finishes, it means all the objects have been classified and the new cluster centre has been calculated. If the value of the evaluation index J does not change after the iteration, the algorithm becomes convergent.

2.7.1 Data generation

Hibench provides the function for the users to produce the k-means data to run their own jobs and it is called *GenKmeansDataset*. The function provides many parameters which relates many aspects of the K-means data. The parameters are lists below:

numSamples: it stands for the amount of the input data and the default value is 20.

numClusters: it represents how many clusters the users want to classifiy and the default value is 2.

meanMin: it shows the min value of the centre ID and the default value is 0.

meanMax: it shows the max value of the centre ID and the default value is 1000.

stdMin: it presents min standard deviation of the clusters and the default value is -100.

stdMax: it presents max standard deviation of the clusters and the default value is 100.

After setting the proper parameters, one can produce the input data. There are two important files need to be produced: sample data and cluster centre. For sample files, the generated data needs to follow the *Gaussian Distribution*. For centre files, the data needs to follow the *Uniform Distribution*[9]. Listing 2.10 and 2.11 shows the sample data and centre data respectively.

```
yjliu@it066427:~/mahout-distribution-0.8$ bin/mahout seqdumper -i /user/yjliu/HiBench/
    Kmeans2/Input/cluster/part-00000
Running on hadoop, using /usr/bin/hadoop and HADOOP_CONF_DIR=
MAHOUT-JOB: /home/yjliu/mahout-distribution-0.8/mahout-examples-0.8-job.jar
19/02/21 15:21:11 INFO common.AbstractJob: Command line arguments: {--endPhase=[2147483647],
    --input=[/user/yjliu/HiBench/Kmeans2/Input/cluster/part-00000], --startPhase=[0], --
    tempDir=[temp]}
Input Path: /user/yjliu/HiBench/Kmeans2/Input/cluster/part-00000
Key class: class org.apache.hadoop.io.Text Value Class: class org.apache.mahout.clustering.
    kmeans.Kluster
Key: CL-0: Value: CL-0{n=0 c=[991.089, 311.404, 479.975] r=[]}
Key: CL-1: Value: CL-1{n=0 c=[323.392, 230.470, 948.572] r=[]}
Key: CL-2: Value: CL-2{n=0 c=[93.758, 947.342, 828.972] r=[]}
Key: CL-3: Value: CL-3{n=0 c=[182.601, 717.132, 629.741] r=[]}
Key: CL-4: Value: CL-4{n=0 c=[737.270, 576.215, 476.047] r=[]}
Count: 5
19/02/21 15:21:12 INFO driver.MahoutDriver: Program took 1385 ms (Minutes:
    0.023083333333333334)
```

Listing 2.10: Cluster center

```
Key: 5940: Value: {0:248.1922426083087,1:615.8217933684431,2:242.31481991717317}
Key: 5941: Value: {0:252.86279962453887,1:636.6988427112975,2:325.85848413896775}
Key: 5942: Value: {0:281.2925681567665,1:650.2565965933796,2:261.12539678482216}
Key: 5943: Value: {0:293.4406039488976,1:639.2207704638387,2:230.5379226579626}
Key: 5944: Value: {0:278.6631414196989,1:646.7933209171147,2:253.48443687107198}
Key: 5945: Value: {0:335.5188483203682,1:607.2100396828683,2:391.95039254788986}
Key: 5946: Value: {0:251.6259457244803,1:640.0831933869397,2:378.04124725224}
Key: 5947: Value: {0:197.9698231486379,1:583.9066387893334,2:344.93375432980946}
Key: 5948: Value: {0:307.549695068334,1:613.3642987144576,2:396.10896707109805}
Key: 5949: Value: {0:283.5729689416751,1:671.5237069686058,2:388.1575519127281}
Key: 5950: Value: {0:269.25119999416575,1:599.53061357693,2:217.2809952160484}
Key: 5951: Value: {0:323.5568499592137,1:651.4818861498582,2:295.718773387541}
Key: 5952: Value: {0:254.40937702957706,1:618.9342329663795,2:210.382983684450 12}
Key: 5953: Value: {0:214.70868416753376,1:638.9056014157164,2:248.00265795206633}
Key: 5954: Value: {0:213.3484564659689,1:636.7858086235894,2:240.85553259823314}
Key: 5955: Value: {0:340.6268314446697,1:632.8695608143396,2:214.5867309076366}
Key: 5956: Value: {0:178.92283591617422,1:626.6015313258697,2:268.08134931697407}
Key: 5957: Value: {0:253.1845845279675,1:612.3703939264973,2:323.25573799595225}
Key: 5958: Value: {0:286.5223123874617,1:591.0367435252917,2:224.4974411201224}
Key: 5959: Value: {0:205.959273589665 88,1:549.2507270487531,2:227.46023805989003}
Key: 5960: Value: {0:204.6780739447981,1:627.8053573165191,2:335.105018051255 74}
Key: 5961: Value: {0:276.6531239489219,1:674.2753486379368,2:347.842935513984 74}
Key: 5962: Value: {0:189.2760523170216,1:641.4104863230958,2:374.887322043438 25}
Key: 5963: Value: {0:296.96727412080463,1:618.0205373080661,2:403.51705280535975}
Key: 5964: Value: {0:259.3588370212758,1:618.8978183303641,2:280.9729888465}
Key: 5965: Value: {0:215.3465459324441,1:582.2264066695303,2:303.75075172094864}
Key: 5966: Value: {0:374.3269303547928,1:615.984176069731,2:385.17288838608977}
Key: 5967: Value: {0:289.19639353812397,1:646.9296020064971,2:238.5140768661576}
Key: 5968: Value: {0:264.9783793404725,1:614.2511229761428,2:348.357398082668}
Key: 5969: Value: {0:275.2356508338942,1:619.3423303737349,2:218.23778858912664}
Key: 5970: Value: {0:256.1867714015175,1:580.2176654467804,2:335.38920106696577}
Key: 5971: Value: {0:314.1784756173183,1:650.6208210041889,2:225.29679223611447}
Key: 5972: Value: {0:314.4491574092808,1:629.8961667745092,2:347.85410317139826}
Key: 5973: Value: {0:286.4745621619131,1:620.022397279952,2:385.3554510462057}
Key: 5974: Value: {0:294.8780255558405,1:638.7721883722536,2:270.5765971265742}
Key: 5975: Value: {0:180.55762223204903,1:618.6712686526174,2:260.246144056911 13}
Key: 5976: Value: {0:306.349691445321,1:594.1896034534727,2:353.9238467604261}
Key: 5977: Value: {0:269.23505604732486,1:618.5808188161047,2:304.1027055821388}
Key: 5978: Value: {0:245.6371895307878,1:668.454043899714,2:332.73126371190887}
```

[9]Another form of normal distribution.

```
Key: 5979: Value: {0:269.49518074971627,1:628.3886385011664,2:439.23234290191533}
Key: 5980: Value: {0:277.24822744330703,1:629.0758099826897,2:311.09850030366346}
Key: 5981: Value: {0:335.11411545229316,1:642.1727253180827,2:212.37587821024445}
Key: 5982: Value: {0:225.88870686847144,1:653.4045487080817,2:301.1580371877249}
Key: 5983: Value: {0:290.29913754002706,1:644.2626645224626,2:215.82184676579413}
Key: 5984: Value: {0:325.23209393030953,1:637.3078474588388,2:209.38125180730466}
Key: 5985: Value: {0:315.3899837786172,1:634.5974432362508,2:242.11845837486405}
Key: 5986: Value: {0:281.5353545114224,1:644.9632205069469,2:325.068826826837O4}
Key: 5987: Value: {0:383.8392911622615,1:613.3829076622408,2:262.15336610729423}
Key: 5988: Value: {0:288.1821924353483,1:620.7446307926359,2:386.68666478028285}
Key: 5989: Value: {0:205.43291353107287,1:631.9604793003757,2:256.68620370381745}
Key: 5990: Value: {0:276.9658511612721,1:646.026781540706,2:280.99288618550554}
Key: 5991: Value: {0:234.11753816918804,1:666.7659250101646,2:379.605837528803}
Key: 5992: Value: {0:324.3154814499773,1:673.8340099890765,2:360.4039815482136}
Key: 5993: Value: {0:237.641361764682l6,1:617.7773528655733,2:397.5634563930S3}
Key: 5994: Value: {0:277.53690926335076,1:618.500961337162,2:406.83658711577164}
Key: 5995: Value: {0:302.2163149452197,1:694.7452469586228,2:335.8382690167798}
Key: 5996: Value: {0:351.031927119715,1:583.5618142373596,2:357.3704645136096}
Key: 5997: Value: {0:307.81843027210675,1:631.6678205210159,2:393.07607100445364}
Key: 5998: Value: {0:307.1831210025599,1:571.4964221593333,2:268.9166116308314}
Key: 5999: Value: {0:316.29502004131564,1:613.6815915705238,2:359.02052829047443}
Count: 6000
19/02/21 15:38:02 INFO driver.MahoutDriver: Program took 1597 ms (Minutes:
    0.026616666666666667)
yjliu@it066427:~/mahout-distribution-0.8$
```

Listing 2.11: Sample points

2.7.2 Algorithm explanation

The principle of k-mean [42] is as follows: this algorithm is for dataset D with n objects and K as the initial cluster numbers.

1. Randomly extract k objects from dataset D as cluster centres.
2. According to the centre of the cluster, assign N objects into the most similar cluster. The similarity depends on the distance between centre and the objects.
3. Update the centre of the clusters which means re-calculate the mean of each cluster.
4. Calculate the criterion function.
5. If the result of criterion function meets the threshold, close the process, else repeat the step two.

The criterion function adopts the following two methods: the first one is global error function, the formula is like this: [41]

$$E = \sum_{i=1}^{k} \sum_{Xj \epsilon Si} (Xj - Ui)^2 \tag{2.1}$$

Where E represents the error, K represents the amount of clusters, S_i represents one of the K clusters, U_i represent centre point of S_i, X_j represents the elements of S_i.

The other method is called central error function,the formula is like this: [41]

$$E = \sum_{i=1}^{k} (U_i^b - U_i^a)^2 \tag{2.2}$$

Where E represent the error, K represents the amount of clusters, i represents the cluster number, U_i^b represents the centre point of the previous cluster, U_i^a represents the centre point of the next cluster.

The process of the K-means can be seen from Figure 2.14.

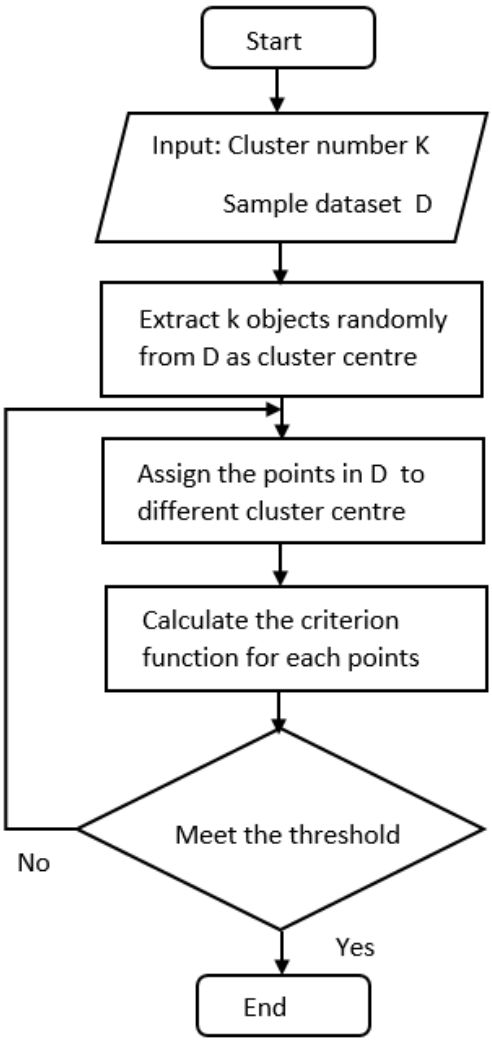

Figure 2.14: Process of K-means modified from [41]

2.7.3 K-means in MapReduce

MapReduce implements K-means algorithm by calling the functions provided by Mahout. Thus, we are going to talk about K-means in Mahout [43]. For Mahout, the K-means algorithm consists of two parts: one is an external loop which will be executed once the criterion function is not meet, the other is an inner loop which is the calculation process of

the algorithm. Mahout adopts *KmeansDriver* to set the loop and utilize *KmeansMapper* and *KmeansReducer* as the main body. The input of the algorithm includes two paths which are the sample data and initial cluster centre vector path. Since Mahout requires the data to be sequencefile format, all the input data should be key-value pairs which key should be configured as *Text* while the value should be *vectorWritable*.

Mahout implements K-means clustering through two steps: Initial division and calculation process [44].

Initial division

Generate k initial partitions in the specified clusters directory and store them in the form of Sequence File. The selection method hopes to avoid isolated points as the centre of Cluster. This step is implemented by *RandomSeedGenerator* class. The process is below:

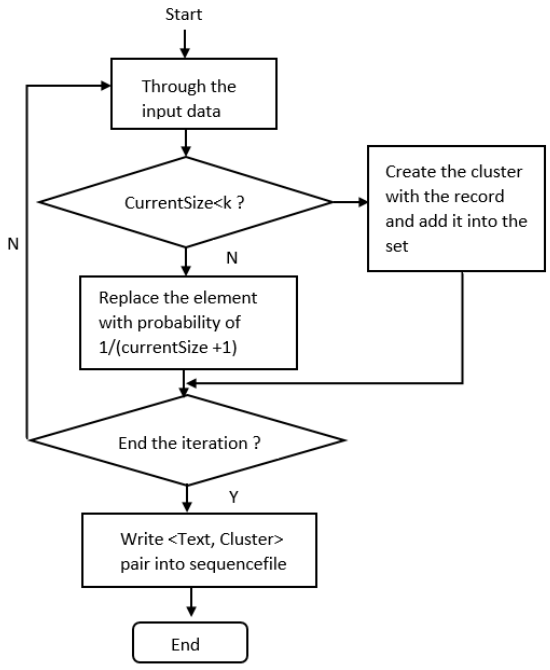

Figure 2.15: Initial division modified from [45]

Calculation step

Calculation step includes two map operations, one combiner operation and one reduce operation. It is trigged by two different jobs and organized by *KmeansDriver*. The execution sequence is as follows:

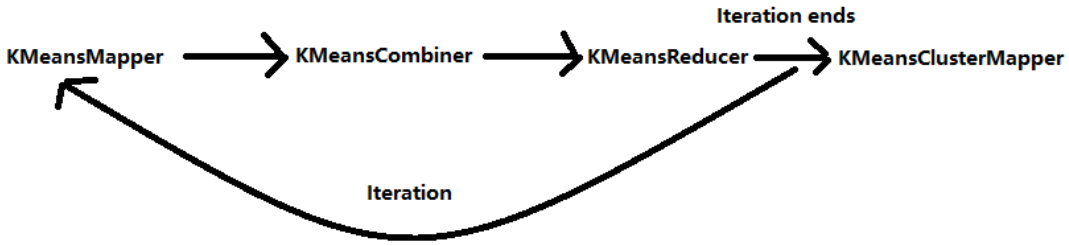

Figure 2.16: Calculate steps modified from [45]

KMeansMapper: firstly, read the cluster centre from last iteration or the initial step. Then each slave utilizes the *emitPointToNearestCluster* method to add each point to the nearest cluster. The output are key-value pairs where the key represents cluster ID while the value represents the instance of *KmeansInfo* which includes the number of points and the points belongs to the cluster.

KMeansCombiner: it is a local reduce operation aimed at merging different component from same cluster ID from *KMeanMapper* output.

KMeansReducer: Accumulating the number of points with same cluster ID and compress all of the points with same ID into the proper key-value pairs. Based on that, calculating the cluster centre for this iteration. If the distance between old cluster centre and the new cluster centre meets the accuracy standard, record the convergence status.

KMeansDriver: Controls the iterative process until the maximum number of iterations is exceeded or all clusters have been convergent. After each iteration, *KMeansDriver* reads all clusters in its clusters-N directory. If all clusters have been convergent, the entire kmeans clustering process becomes convergent.

In addition, the results of each iteration will be saved on the HDFS for further use. When the next iteration needs the previous data, MapReduce will read the previous data from HDFS and then calculate new iteration results.

2.7.4 K-means in Spark

Spark provides the function to implement K-means algorithm and the function is under *MLib* package. The implementation of *Spark.mlib* includes a parallelized variant method to initialize cluster centre which is called kmeans‖ [46][10]. There are some parameters *Spark.mlib* provides for K-Means:

- K: it stands for the amount of the clusters.
- MaxIterations: it represents the maximum iterations to run.
- InitializationMode: it presents how to initialize the centre points of the cluster. Spark provides two methods: Random initialization and k-means‖.

[10] An modified method of k-means.

- Runes: it presents parallelism which means how many K-means algorithm run together under the same settings.
- Epsilon: it presents the distances threshold where we can recognize K-means as converged.

Similar to K-means in Mahout, Spark also implements K-means in two steps: centre initialization and calculation process [47].

For centre initialization, the default method is K-means‖ because random cluster centre may lead to wrong classification results. The principle of K-means‖ [46] are as follow:

1. Select a point randomly from the initial points as the centre of the cluster.
2. For every point from the dataset, calculate the distance D(x) from the cluster centre.
3. Choose a new point as the new cluster centre. The rule of the selection is that: the larger D(x) is the higher probability it will be chose.
4. Repeat step2 and step3 until get K cluster centres.
5. Return all the cluster centres.

For calculation step, Spark receive the cluster centre points from one of the initialization methods. Then it will repeat the next two steps to get the proper K cluster:

- Calculate each distance between the points from the dataset and the centre points of each cluster and then put each point into different clusters depends on the nearest centre points.
- Recalculate the centre points.

After finished initializing the centre points, Spark will broadcast them to each RDD. Then, each RDD will run *mapPartition* operator to calculate the distance between each point and the cluster centres and classify points to different clusters based on the nearest centre point. After that, Spark utilize *reduceByKey* operator to merge the clusters information and *collectAsMap* operator to output the result.

After getting the results, Spark will recalculate the centre points by calculating the arithmetic mean of each component from the getting clusters. If the distance between the previous centre points and the current centre points is bigger than the given *Epsilon*, the next iteration will begin. Otherwise, the K-means iteration ends and get the results.

The results of each iteration will be saved by different persist level. The users can utilize the cache operation to save each iteration results on the memory or on the disks.

2.8 MapReduce optimization strategies

In this section, we describe the optimization strategies for MapReduce jobs. According to Holmes [48], the factors that have a negative effect on job performance can be divided into the following categories:

1. Hadoop Configuration. The default settings under most clusters show low efficiency while the same situation happens if the personalized configuration not set properly. The common performance problems caused by configuration are frequent memory swapping, CPU overload and so on.

2. Map task. Extremely large and small files can affect the performance of Map tasks. Also, poorly managed code can have an impact.

3. Reduce tasks. Data skew and the number of reduce tasks can affect the performance of reduce tasks.

4. Hardware. The performance can be affected by some bad nodes and poor network especially for the small cluster.

Since one and four are unable to be solved by parameter tuning, we are going to talk about what kinds of problems the job will meet during the map and reduce stages and list some parameters that can improve the situation.

Figure 2.17 illustrates the whole process of map tasks and the factors that may affect the job efficiency during the process. There are four stages exists in the map tasks. The first one is the scheduling process and it happens between the job started and the map tasks started. During the process, the queue resource and the number of jobs in the queue are the most serious problems. Based on MapReduce online documents [49], there are several parameters can affect the situation which are: *mapreduce.map.memory.mb* and *mapreduce.map.cpu.vcores*. These two parameters are used for configuring the resource utilization for each map tasks. By tuning these two, the job can be more flexible to apply resource for different sizes of jobs.

The second stage is Read inputs. It happens at the beginning of map tasks. During the process, the most serious problem is the size of the input files. Based on the online documents [49], *mapred.min.split.size* and *mapred.max.split.size* are utilized to control the sizes of input files for each map tasks. Since the default block size for HDFS is 128MB, once the *mapred.min.split.size* bigger than block size, one block will be splits to different pieces and assign to different map tasks. On the contrary, if the *mapred.min.split.size* is smaller than the block size, several blocks of data will be assigned to one map tasks. Thus, by configuring these parameters, the users can easily control the size of input data to improve the efficiency.

The third and fourth stage can be combined together because they are about the map function and the result output. For the map function, no parameters are available to improve the quality of codes. Thus, the fourth stage is more important. From the table, the most serious problem for Spill is local hardware issue. Based on the online document, the result of map function will be saved on the buffer and the buffer will spills as new files once it meets the threshold. MapReduce provides the parameter called *mapreduce.io.sort.mb* to control the sizes of the buffer. Also, after finished spills new files, MapReduce will merge the files together and the parameter called *mapreduce.io.sort.factor* is utilized to control that. In addition, there are many other parameters that may affect the efficiency of map tasks [49]: *mapreduce.map.sort.spill.percent*, *mapreduce.map.output.compress* and *mapreduce.map.output.compress.codec*. Table 2.2 concludes all the parameters for map tasks.

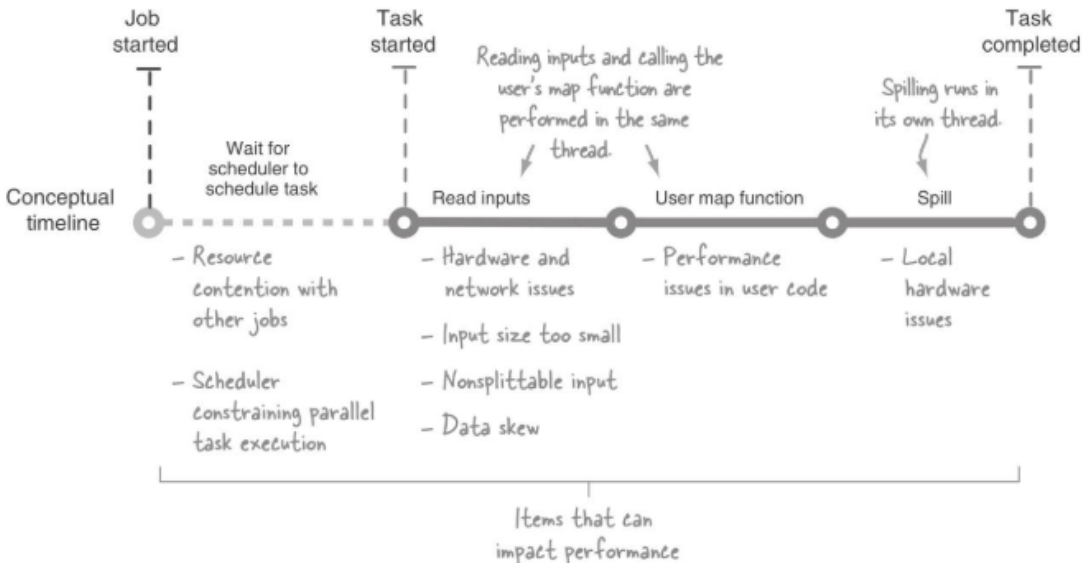

Figure 2.17: The timeline of Map tasks and the impacts on the job efficiency from [48]

Parameters	Type	Default value	Description
mapreduce.map.memory.mb	Int	2048	Memory for each map task
mapreduce.map.cpu.vcores	Int	1	Vcores for each map task
mapred.min.split.size	Int	1	The min value of Input splits
mapred.max.split.size	Int	128	The max value of Input splits
mapreduce.io.sort.mb	Int	100	The buffer for map tasks
mapreduce.io.sort.factor	Int	10	The number of spill files merge together
mapreduce.map.sort.spill.percent	Float	0.7	The threshold of map buffer
mapreduce.map.output.compress	boolean	False	Whether compress the intermediate data for map tasks
mapreduce.map.output.compress.codec	String	Null	Compression algorithm

Table 2.2: The parameters for Map tasks modified from [49]

Figure 2.18 illustrates the timeline of reduce tasks and the factors that may affect

the job efficiency. There are five stages during the whole process. The first stage happenes between job started and the reduce tasks started and the main problem are same to the Map tasks. Based on the online documents, *mapreduce.reduce.cpu.vcores* and *mapreduce.reduce.memory.mb* can be utilized to solve the problem. By configuring the parameters for Reduce tasks, the users can control the resource they need and thus to improve the efficiency.

The next stage is the shuffle process and it happens between reduce tasks started and all inputs read. Since it represents the process to transfer the data from map side to reduce side, the most serious problems are hardware and network issues. MapReduce provides one parameters to improve the job efficiency which is *mapreduce.reduce.shuffle.parallelcopies*. During the shuffle process, the number of map results are more than one and this parameter can be utilized to pull more results for one time to improve the efficiency. Also, since the files downloaded from map tasks need to be merged, *mapreduce.io.sort.mb* can be utilized to control the merging process.

Since there are not any parameters prepared for sort and reduce process, we combine the last three stages together. There are many parameters prepared for these three stages which are *mapred.job.shuffle.input.buffer.percent*, *mapred.job.shuffle.merge.percent* and *mapred.job.reduce.input.buffer.percent* [49]. Table 2.3 presents all the parameters for Reduce tasks.

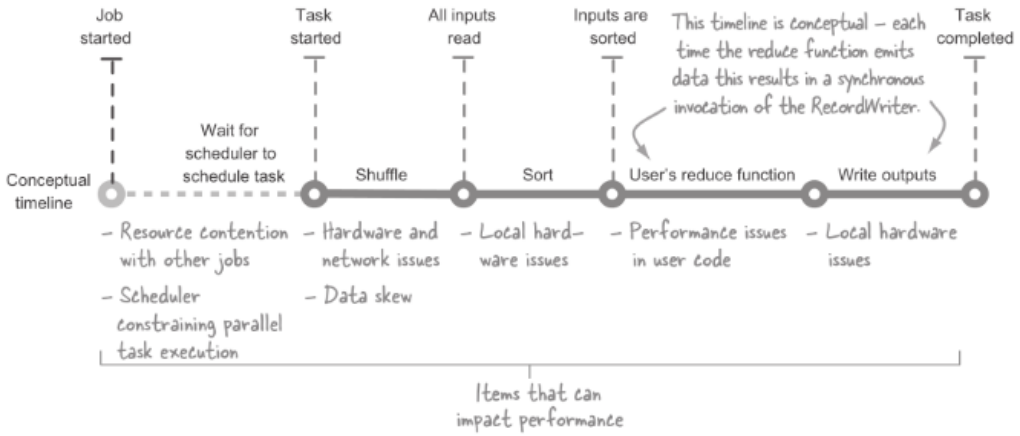

Figure 2.18: The timeline of Reduce tasks and the impacts on the job efficiency from [48]

Parameters	Type	Default value	Description
mapreduce.reduce.memory.mb	Int	8192	Memory for each reduce task
mapreduce.reduce.cpu.vcores	Int	1	Vcores for each reduce task
mapred.reduce.parallel.copies	Int	5	The number threads used to download the map resultss
mapreduce.io.sort.factor	Int	10	The number of spill files merge together
mapred.job.shuffle.input.buffer.percent	float	0.7	The ratio of buffer size takes for the reduce tasks
mapred.job.shuffle.merge.percent	float	0.66	The threshold when the spill happens
mapred.job.reduce.input.buffer.percent	float	0.0	How much memory is used to store data in the buffer

Table 2.3: The parameters for Reduce tasks modified from [49]

2.9 Spark optimization strategies

Regarding to the optimization for Spark, there are many suggestions from the Internet. According to Karau [50], the basic tuning strategy for Spark is configuring the parameters related to Spark core. Also, Spark online documents propose other aspects for tuning Spark jobs that includes the shuffle process, internet problem and memory management. Thus, we are going to talk about the two sides and propose the parameters that may related to the process.

As we explained how Spark works from the last chapter, we are going directly to the parameters provided by Spark core to control the jobs. There are 7 parameters provided by Spark to set the jobs that are *num-executors*, *executor-memory*, *executor-cores*, *driver-memory*, *spark.default.parallelism*,*spark.storage.memory.Fraction* and *spark.shuffle.memory Fraction*. Next, we are going to introduce all of them briefly and give the suggestions about how to decide these parameters.

num-executors

This parameter is used to configure how many executors launch for Spark jobs. When the driver program applies the resource from the resource manager, the resource manager will launch the corresponding number of executors on each working node as many as possible to meet the requirement [51]. This parameter is the most important parameter that must be set properly. Since the default settings will only launch small number of executors leading the job run slowly.

The suggested number of executors are between 50 and 100. Since the large number of executors may take too much queue resource leading other jobs has little resource to

apply while the small number of executors may not fully utilize the queue resource, thus the values in this range is strongly suggested.

executor-memory

This parameter is utilized to configure the memory for each executor. Since memory is the most important resource for Spark jobs and may affect the efficiency directly, this parameter must be configured properly. Also, the problem related to *JVM OOM* is always caused by this setting.

The proper setting should around 4GB to 8GB for each executor. However, the specific values should depend on the size of the cluster. Also, if there are more than one jobs running on the queue, the total memory should not exceed 1/3 or 1/2 of the cluster resource to ensure other jobs can run smoothly.

executor-cores

This parameter is utilized for configuring the number of CPU cores for each executor. This parameter determines the parallel computing ability for each executor. Since one CPU core can execute many tasks at a time, the more CPU cores assigned to the executors, the faster the Spark jobs can be.

The proper settings should around 2 to 4 for each executor. Also, the specific values should depend on the requirement of the jobs. Besides, the total CPU cores should not exceed 1/3 or 1/2 of the cluster resource to ensure other jobs can run smoothly.

driver-memory

This parameter is used to set the memory for driver program.

This parameter is optional and not configured under some circumstances. The only thing needs to be care is that the value needs to be high enough if the jobs include the *collect* operation to gather all the data to the driver process.

spark.default.parallelism

This parameter is used to configure the default amount of tasks for each stage [52]. This parameter can affect the efficiency of the Spark jobs significantly once it is configured properly. Since one partition correspond to one task, this parameter is actually to set the parallelism of tasks.

The proper setting should around 500 to1000. If this parameter is not configured, the default amount of tasks will be decided by the blocks of HDFS and one block correspond to one task. Normally, this default setting is smaller than expected which means the few tasks takes large resource. Thus, the efficiency will be low and the resource assigned to

executors will be wasted. The suggestion is that setting this parameter twice or three times as many as *num-executors* * *executor-cores* to fully utilize the resource.

spark.storage.memory.Fraction

This parameter is utilized for setting the percentage of executor memory which can be utilized for *cache* operation and the default value is 0.6. In other words, 60% of the executor memory can be utilized to save the RDDs. Based on the different RDD persist level, this parameter can help the user save the data on local disks or memory.

If there are many *cache* operations in Spark jobs, the value should be configured larger to ensure all the data the user needs can be saved on the memory. On the contrary, if there are many *shuffle* operations in Spark jobs, the value will be configured smaller to ensure the efficiency.

spark.shuffle.memoryFraction

This parameter is used to set the percentage of executor memory that can be used for *aggregation* operations after a task pulled the output from the previous stage during the *shuffle process*. The default value is 0.2 [51]. In other words, 20% of the memory can be utilized for this operation. If the memory is above the value during the *shuffle* process, Spark will write the other data into the local files and thus reduce the efficiency.

If there are many *cache* operations and few *shuffle* operations in the Spark jobs, the value should be configured smaller to avoid the efficiency lose because of too much data during the *shuffle* process.

Apart from the optimization for Spark core, Spark provides many other parameters that related to other aspects of the jobs. Table 2.4 and Table 2.5 illustrates some important parameters related to the internet problems as well as the *shuffle* process respectively.

Parameters	Type	Default value	Description
spark.rpc.message.maxSize	Int	128	The largest message size allowed in "control plane" communication. It only applies to describe the size of information sent between executors and the driver
spark.network.timeout	Int	120	The default timeout for all the network application
spark.port.maxRetries	Int	16	Maximum number of times when bind to a port
spark.rpc.lookupTimeout	Int	120	The duration for an RPC remote endpoint operation waiting before time out

Table 2.4: The parameters for internet problems modified from [20]

Parameters	Type	Default value	Description
spark.shuffle.file.buffer	Int	32	Buffer of Bufferoutputstream during the shufflewriter process
spark.reducer.maxSizeInFlight	Int	48	Buffer size of shuffle read tasks which controls how much data can be pulled for once
spark.shuffle.manager	String	Sort	The type of shuffle manager. There are three options: hash, sort and tungsten-sort
spark.shuffle.sort.bypassMergeThreshold	Int	200	When the shuffle manager is sort, if the number of shuffle read tasks is smaller than the threshold, shuffle writer will not perform sort operation while write the data based on Hash values.
spark.shuffle.consolidateFiles	Boolean	false	This parameter will be valid if the user choose HashShuffleManager. If the value is true, the consolidate mechanism will be enabled and the output of the shuffle writer will be merged.

Table 2.5: The parameters for shuffle performance modified from [20]

2.10 Summary

In this chapter many interesting literatures have been discussed to evaluate the performance of the cluster. Based on them, Hadoop and Spark are the most powerful and popular frameworks that can put the ideas of process large volume of data into practice. Also, Hibench suite provides us many interesting workloads to evaluate the performance of the cluster from many aspects. Thus, we decide to conduct our own experiments to test the performance of our cluster with the three existing jobs in HiBench suite.

Since the limitation of HiBench suite is obvious, we decided to withdraw three workloads from HiBench suite and run the jobs on our own cluster. We are going to implement all the workloads by Spark and MapReduce. This can help us evaluate the performance of the jobs through more aspects. In addition, apart from implementing the jobs, we decide to put parameters tuning into consideration. By configuring the jobs through many aspects, we hope to see the difference between Spark and MapReduce and how much the parameters can affect the job efficiency. Thus, we choose Yarn as our Resource Manager because it provides a set of complete system to monitor the cluster performance while giving us freedom to tune different parameters related to MapReduce and Spark.

Chapter 3

Experiment settings

3.1 Hardware configuration

Our Hadoop and Spark clusters are built under the same hardware configuration with 10 nodes. Each node has 8 CPU cores at 2.9 GHz, 1 Tb disk, and 32 GB physical memory. Overall, our 10 nodes cluster owns 104 CPU cores, 420 GB RAM and 51 TB local storage. Regarding to its reading ability, our cluster provides an aggregate bandwidth of about 10 GB/second. Also, the writing speed is about 5 GB/sec through dd test. Our nodes are connected by 1,000 MB internet and run Ubuntu 16.04.2 (GNU/Linux 4.13.0-37-generic x86_64).

As a comparison, the hardware of our cluster is roughly equivalent to a cluster with 40 virtual machines. Our hardware is suitable for handling various difficult situation in Spark and Hadoop. For example, there are many concurrent tasks running on our cluster and sometimes more than one type of jobs exists on the cluster which prove that our cluster has enough CPU cores and RAM to solve any kinds of problems. However, experiments related to large number of nodes (e.g., evaluating the scalability of different nodes) are out of the scope of our thesis.

3.2 Software configuration

Both Spark and Hadoop are configured on Java 1.7.0.

Hadoop: we choose Hadoop version 2.4.0 to run MapReduce job and selected Yarn as the resource manager. All of the 10 nodes can be used to sort the data and the intermediate data will be saved on HDFS. We keep the default settings of HDFS which utilize 128MB as block size and 3 as replication factor. In order to control the parallelism degree of the job better, we enable the CPU-scheduling function on YARN. Also, we give each map tasks 7 GB memory for map tasks and 14GB memory for reduce tasks.

Spark: we use Spark 2.1.0 to run the jobs on YARN. The intermediate data is also

saved on HDFS. We set the default number of executors as two and each of the executor own 1 vocre and 1GB memory.

3.3 Profiling tools

In this section, we present a novel and powerful cluster management tool which we have used to monitor and profile the selective workloads running on Spark and Hadoop. It is called Apache Ambari.

3.3.1 Ambari

Apache Ambari is a web-based project aimed at simplifying Hadoop management. Until now, it already supports most of the Hadoop components, "including HDFS, MapReduce, Hive, Pig, Hbase, Zookeeper, Sqoop, and Hcatalog" [53]. In addition, Ambari supports the users to control the Hadoop cluster on the following three aspects [53]:

Provision

- Ambari provides detailed instruction for configuring Hadoop cluster to many nodes.
- Ambari integrates all the important configuration settings related to Hadoop cluster on the website.

Management

- Ambari provides powerful management system which includes start, stop and re-configure the Hadoop service across the all the nodes.

Monitoring

- Ambari provides a dashboard to monitor the condition of the whole cluster.
- "Ambari adopts Ambari Metrics System to collect the metric information" [54].
- "Ambari adopts Ambari Alert Framework for system alerting and will remind you when needed (e.g., data node goes down, low disk space, etc)" [54].

Ambari has achieved the following:

1. Simplified cluster provisioning with "a step-by-step installation wizard" [53].
2. Pre-configure key metrics to check whether Hadoop Core (HDFS and MapReduce) and related components (such as HBase, Hive, and HCatalog) are healthy [53].

3. Check the job dependencies and performance by supporting visualize and analysis job execution.

4. The information is exposed through a complete RESTful[1] API that integrates existing operations tools.

5. The user interface is intuitive and allows users to view information and control the cluster easily.

Our experiments include the following components in Ambari: HDFS, MapReduce 2.0, Spark 2.0, Yarn and Mahout.

3.3.2 HDFS:

For our experiment, we use HDFS to upload and downloads important files which includes the input data, Jar package or configuration files. Since the sizes of our data are huge and some of them can reach to 500 GB, HDFS presents its strong storage ability by splitting the data into different blocks and save them on the different data nodes. Also, HDFS not only allows users to save the big files on HDFS, but also it gives us the freedom to visit the HDFS from different locations which enhance its availability. Besides, HDFS supports us conduct reading and writing operation which means when we run our MapReduce or Spark jobs on the cluster, we can select the files on HDFS as the input data or select somewhere as the output directory. In addition, the log files from different nodes will be saved on the HDFS automatically which can help us diagnosis the problem happened during the job running. Finally, Ambari provides multiple metrics to visualize the resource utilization on HDFS. We can monitor the situation of each data nodes by checking the *Heatmaps* function. There are eight indexes integrated under *Heatmaps* function which are: *HDFS Bytes Written*, *DataNode Garbage Collection Time*, *DataNode JVM Heap Memory Used*, *DataNode JVM Heap Memory Committed*, *DataNode Process Disk I/O Utilization*, *DataNode Process Network I/O Utilization*, *HDFS Space Utilization* and *HDFS Bytes Read*. These functions are powerful enough to help us understand the situation of each datanode under working and idle situation.

3.3.3 MapReduce 2.0:

MapReduce 2.0 separates "what to do" and "how to do" through abstract model and computing framework, providing the programmers a high-level programming interface and framework. Compared to MapReduce framework, more powerful tools are proposed that include ResourceManger,NodeManger and ApplicationMaster. By splitting the data to be processed by our job and assigning the blocks to different tasks, it achieves distributive

[1] A kind of design strategy which provides a set of design principles and constrain conditions.

computing through different nodes. Thus, the framework can improve the efficiency of our experiment dramatically. Also, once the job finishes, MapReduce can provides the details of the map and reduce stage to help us understand what happened under different nodes. Besides, by tuning the parameters provided by MapReduce, we can improve our job efficiency while reducing the pressure of the cluster.

3.3.4 Spark 2.0:

Similar to MapReduce, Spark 2.0 proposes another distributed computing framework which implements parallel computing by processing different RDDs. Compared to Spark 1.0, the performance improves sharply and the API are more easy to use. For our experiment, the DAG scheduler will split our jobs into different stages and assign different tasks to each stage. Also, by applying different storage strategy, Spark allows us to move the intermediate data from disk to memory and thus improve the job efficiency obviously. Finally, Spark 2.0 provides a set of well-established monitor system which can help us track the situation of different stages or tasks easily.

3.3.5 Yarn:

Essentially, Yarn plays a role of resource manager which applying resources for MapReduce and Spark jobs. This entity controls the entire cluster and manages the allocation of applications to the underlying fundamental resources. Yarn carefully arranges the various resources (calculation, memory, bandwidth, etc.) to the NodeManager and monitor their applications with it. For our experiment, Yarn is mainly charged for monitoring different jobs. All the information related to the jobs can be found from Yarn which includes the parameter settings and job status. Also, Yarn provides the different metrics for resource utilization for the entire cluster. By monitoring the resource utilization of the whole cluster and different nodes, we can evaluate our cluster performance and optimize our jobs from each node.

3.3.6 Mahout:

Mahout is a unique member of the Hadoop ecosystem, it is a distributed computing framework based on machine learning and data mining. It not only provides the implementation of many professional machine-learning algorithms, but also extends them into the Hadoop clusters. For our project, we use Mahout to pre-process the input data and implement K-means algorithm based on that. Also, since the result are also under sequencefile format, Mahout can help us transfer the files into text format for further analysis.

3.4 Experiment workflow

In this section, we are going to present the experiment workflow and how to use the tools we mentioned above to conduct the experiment. Also, this part includes two steps: data preparation and workloads execution.

3.4.1 Data preparation

Before putting the workloads into practice, we need to produce the input data. We utilize the function provided by HiBench suite to create the input data. Based on the package, the first step is finding the directory of the specific workloads and set the size of the jobs we want. Listing 3.1 presents how we set different sizes of datasets for Wordcount.

```
#datagen
hibench.wordcount.tiny.datasize                 32000
hibench.wordcount.small.datasize                320000000
hibench.wordcount.large.datasize                3200000000
hibench.wordcount.huge.datasize                 32000000000
hibench.wordcount.gigantic.datasize             320000000000
hibench.wordcount.bigdata.datasize              1600000000000
hibench.wordcount.10g.datasize                  10459610972
hibench.wordcount.50g.datasize                  52298054860
hibench.wordcount.100g.datasize                 104596109720
hibench.wordcount.150g.datasize                 156894164580
hibench.wordcount.200g.datasize                 209192219440
hibench.wordcount.250g.datasize                 261490274300
hibench.wordcount.300g.datasize                 313788329160
hibench.wordcount.350g.datasize                 366086384020
hibench.wordcount.400g.datasize                 418384438880
hibench.wordcount.450g.datasize                 470682493740
hibench.wordcount.500g.datasize                 522980548600


hibench.workload.datasize               ${hibench.wordcount.${hibench.scale.profile}.
    datasize}

# export for shell script
hibench.workload.input                  ${hibench.hdfs.data.dir}/Wordcount/Input
hibench.workload.output                 ${hibench.hdfs.data.dir}/Wordcount/Output
```

Listing 3.1: The configuration of Wordcount modified from [27]

Next, we need to change the settings of HiBench configuration files to let the workloads accept and produce the datasets. Here, we need to go the directory of *conf* files under HiBench suite and change the settings of *hibench.scale.profile* based on the size of datasets we want. Listing 3.2 presents the situation that we want to produce the 50GB data for wordcount.

```
Data scale profile. Available value is tiny, small, large, huge, gigantic and bigdata.
# The definition of these profiles can be found in the workload conf file i.e. conf/
    workloads/micro/wordcount.conf

hibench.scale.profile                   50g
# Mapper number in hadoop, partition number in Spark
hibench.default.map.parallelism         10

# Reducer nubmer in hadoop, shuffle partition number in Spark
hibench.default.shuffle.parallelism     25


```

```
12 #================================================================
```

Listing 3.2: The configuration of Wordcount modified from [27]

After setting all the configurations properly, we are going to produce the input data. The preparation function is under the bin directories and we can select any workloads we want and create the input data with the prepare function. Listing 3.3 and Figure 3.4 show how we create the input data for Wordcount and the files on HDFS respectively.

```
1  yjliu@it066427:~/Hibench/HiBench-master/bin/workloads/micro/wordcount/prepare$ ./prepare.sh
2  patching args=
3  Parsing conf: /home/yjliu/Hibench/HiBench-master/conf/hadoop.conf
4  Parsing conf: /home/yjliu/Hibench/HiBench-master/conf/hibench.conf
5  Parsing conf: /home/yjliu/Hibench/HiBench-master/conf/spark.conf
6  Parsing conf: /home/yjliu/Hibench/HiBench-master/conf/workloads/micro/wordcount.conf
7  probe sleep jar: /usr/hdp/current/hadoop-client/../hadoop-mapreduce/hadoop-mapreduce-client-
       jobclient-tests.jar
8  start HadoopPrepareWordcount bench
9  hdfs rm -r: /usr/hdp/current/hadoop-client/bin/hadoop --config /usr/hdp/current/hadoop-
       client/etc/hadoop fs -rm -r -skipTrash hdfs://it066427:8020//user/yjliu//HiBench/
       Wordcount/Input
10 Deleted hdfs://it066427:8020/user/yjliu/HiBench/Wordcount/Input
11 Submit MapReduce Job: /usr/hdp/current/hadoop-client/bin/hadoop --config /usr/hdp/current/
       hadoop-client/etc/hadoop jar /usr/hdp/current/hadoop-client/../hadoop-mapreduce/hadoop-
       mapreduce-examples.jar randomtextwriter -D mapreduce.randomtextwriter.totalbytes=32000 -
       D mapreduce.randomtextwriter.bytespermap=3200 -D mapreduce.job.maps=10 -D mapreduce.job.
       reduces=20 hdfs://it066427:8020//user/yjliu//HiBench/Wordcount/Input
12 ECDSA key fingerprint is SHA256:jzLxe64jvR403rWO6I2AoSnKZsi+pip1IWE+cC2OZtc.
13 The job took 24 seconds.
14 finish HadoopPrepareWordcount bench
15 yjliu@it066427:~/Hibench/HiBench-master/bin/workloads/micro/wordcount/prepare$
```

Listing 3.3: The process of creating the input data for Wordcount

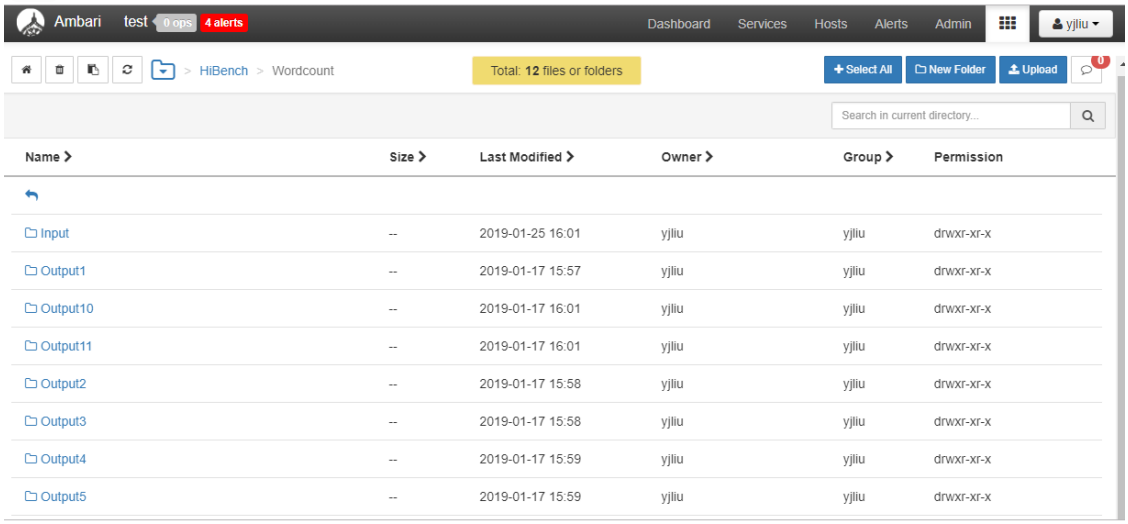

Figure 3.1: wordcount files on HDFS

The last step is to check whether the input data is under correct formats for our experiment. Since the input data is under sequencefile format, we are unable to open it

directly to check the content. Thus, Mahout is used to help us transform the dataset to make it visible. Listing 3.4 shows the datasets we create for Wordcount.

```
1  yjliu@it066427:~/mahout-distribution-0.8$ bin/mahout seqdumper -i /user/yjliu/HiBench/
       Wordcount/Input/part-m-00000
2  Running on hadoop, using /usr/bin/hadoop and HADOOP_CONF_DIR=
3  MAHOUT-JOB: /home/yjliu/mahout-distribution-0.8/mahout-examples-0.8-job.jar
4  19/02/21 15:33:40 INFO common.AbstractJob: Command line arguments: {--endPhase=[2147483647],
       --input=[/user/yjliu/HiBench/Wordcount/Input/part-m-00000], --startPhase=[0], --tempDir
       =[temp]}
5  Input Path: /user/yjliu/HiBench/Wordcount/Input/part-m-00000
6  Key class: class org.apache.hadoop.io.Text Value Class: class org.apache.hadoop.io.Text
7  Key: acocotl overcrown mericarp uncompromisingly drome bromate stachyuraceous Hysterocarpus
       allotropic : Value: sequacity reciprocation unchatteled spermaphyte sedentariness anta
       homotransplant uncompromisingly ethnocracy tomorrowness lyrebird bladderwort flatman
       trip decardinalizesymbiogenetically absvolt atlantite subfoliar ribaldrous unscourged
       oratorize bromic devilwise friarhood abthainry coracomandibular epidymides subangulated
       migrainoid reappreciate ununiformly groundneedle pentosuria supermarket subirrigate
       frenal blightbird phytonic
8  Key: commandingness times barkometer liquidity coracomandibular abstractionism seeingness
       authorling trip : Value: phallaceous ambitus figureheadship unrepealably lebensraum
       rehabilitative various posterishness kenno metaphrastical astucious omniscribent
       approbation tonsure mustafina subofficer Triconodonta cubby culm seraphism Shiah slait
       perculsive barkometer uninhabitedness putative archesporial insatiately Animalivora
       quintette rizzomed astucious sviatonosite orgiastic Joachimite prescriber eristically
       Macraucheniidae warlike rainproof plerome dialoguer equiconvex oinomancy redescend quad
9  Key: circular unchatteled ascitic stormy danseuse tonsure Dunlop : Value: mastication
       overcrown lithotresis swoony almud hemimelus lithotresis archesporial dehairer
       venialness paranephros calycular trailmaking impressor sesquiquintile unswanlike
       iniquitously atlantite zenick mangonism diplomatize Effie valvula raphis trip flatman
       seeingness parmelioid isopelletierin ribaldrous quadrennial licitness bugre brutism
       trabecular octogynous sapphiric laurinoxylon bugre enhedge mendacity mammonish Helvidian
        shellworker meloplasty acocotl steprelationship groundneedle foursquare Jerusalem
       Pishquow erlking embryotic sloped eternal hysterolysis dunkadoo isopelletierin chooser
       taurocolla oblongly noreast pentosuria ten abstractionism mammonish Bermudian
       lophotrichic pope Passiflorales soorkee ribaldrous floatability mechanist
       uninhabitedness slait counteractively
10 Key: Hydrangea choralcelo psychofugal sialadenitis tum : Value: oflete posterishness
       cobeliever pseudohalogen unschematized Mormyrus Munychian diopside boor overcontribute
       guitarist seeingness widdle Orbitolina electrotechnics craglike iniquitously equiconvex
       outwealth arrowworm Lentibulariaceae dastardliness elemicin ticktick karyological naught
        biventer Bushongo lineamental groundneedle arval commotion oinomancy redesertion
       Bulanda entame pope oversand pamphlet vesperal flutist pomiferous topline sequestrum
       sportswomanship unpeople Ophiosaurus slangy returnability ordinant critically
       psychofugal monogoneutic basto subtransverse comism antalgol bot rebilling Pishquow
       penult Glecoma squit scyphostoma silicize craglike decidable appetible reciprocation
       bromic predebit comparability stiffish comism inventurous peptonate elemicin
       orchiocatabasis unefficient idiotize pamphlet Alethea naught Aktistetae bromic
       homotransplant paradisean sheepskin perculsive superindifference Confervales
       unreprimanded Isokontae dishpan raphis nonlustrous flutist culm
11 Key: bespin macropterous bismuthiferous prolificy frictionlessly coldfinch adscendent
       inferent tautness : Value: digitule Alethea chalcites skyshine wingable manilla
       alveolite arsenide seelful oflete tonsure emir unimmortal subtransverse unexplicit
       yawler biopsic unexplicit stereotypography overcrown precostal
12 Count: 5
13 19/02/21 15:33:41 INFO driver.MahoutDriver: Program took 1267 ms (Minutes:
       0.021116666666666666)
14 yjliu@it066427:~/mahout-distribution-0.8$
```

Listing 3.4: The input data under Mahout

3.4.2 Workloads execution

The first step is *ssh* the command line of the master node. There are many software that provides the function. Among them, we select putty and access our cluster with its IP address. The settings of our putty and the command line of our master node can be seen

from Figure 3.2 and Figure 3.3.

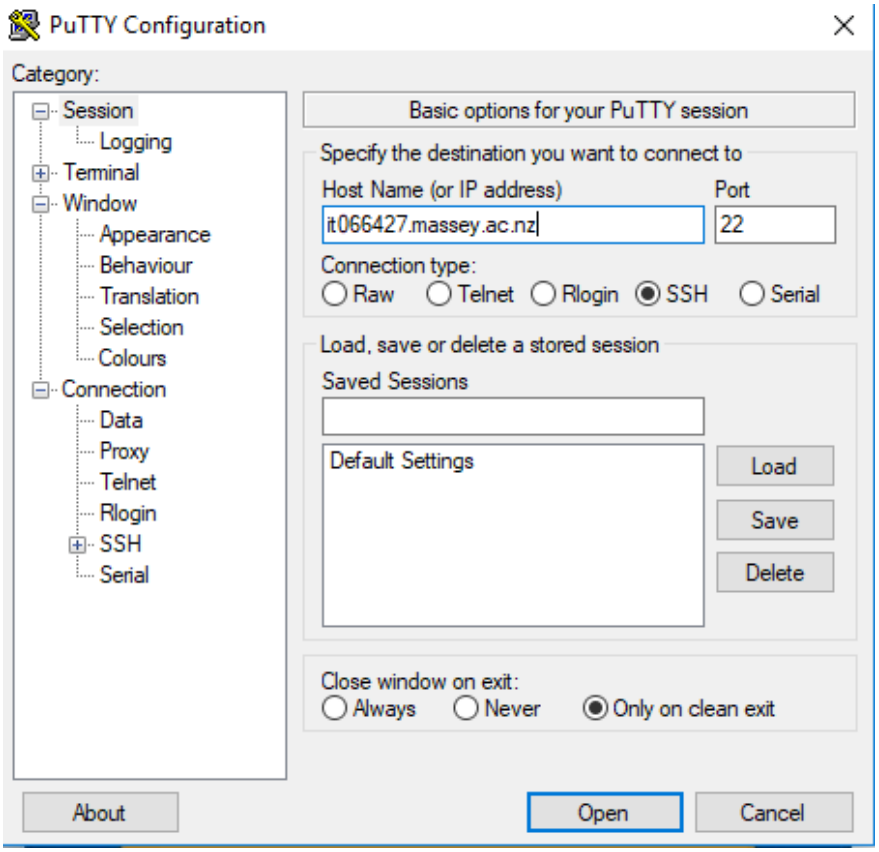

Figure 3.2: The settings of PuTTY

```
yjliu@it066427: ~
login as: yjliu
yjliu@it066427.massey.ac.nz's password:
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.13.0-45-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

301 packages can be updated.
4 updates are security updates.

New release '18.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

*** System restart required ***
Last login: Fri Feb 15 19:56:18 2019 from 130.123.244.74
yjliu@it066427:~$
```

Figure 3.3: The command line of Master node

The second step is settings the environment parameters for MapReduce and Spark. Since our experiments utilize the jar package provided by HiBench, some jobs need to read the configuration files from HiBench directory. Thus, we configure two environment pa-

rameters which are *SPARKBENCH_PROPERTIES_FILES* and *HADOOP_CONF_DIR*. The ways we export these two settings are shown from Listing 3.5 and 3.6.

```
export SPARKBENCH_PROPERTIES_FILES=/home/yjliu/Hibench/HiBench-master/report/terasort/spark/conf/sparkbench/sparkbench.conf
```

Listing 3.5: Export SPARKBENCH_PROPERTIES_FILES

```
export HADOOP_CONF_DIR=/usr/hdp/current/hadoop-client/etc/hadoop
```

Listing 3.6: Export HADOOP_CONF_DIR

The third step is about executing our experiments with the default settings. We need to provide the parameters we want to use and execute the experiment with the correct script. Listing 3.7 presents the example of MapReduce WordCount.

```
/usr/hdp/current/hadoop-client/bin/hadoop jar /usr/hdp/current/hadoop-client/../hadoop-mapreduce/hadoop-mapreduce-examples.jar wordcount hdfs://it066427:8020//user/yjliu//HiBench/wordcount50g/Input hdfs://it066427:8020//user/yjliu//HiBench/Wordcount/Output
```

Listing 3.7: Execute WordCount job in MapReduce

Next, we need to monitor the status of our jobs through Ambari YARN web UI. It provides many useful information including the execution time, resource utilization and tasks status. We can find as much as information as we want from the web UI. Besides, this UI also provide configuration function which allows us to check the parameters setting for the jobs. This function is powerful to help us understand whether the customized parameters settings works or not. The Figure 3.4, 3.5 and 3.6 show the Ambari UI, Yarn UI and Configuration function respectively.

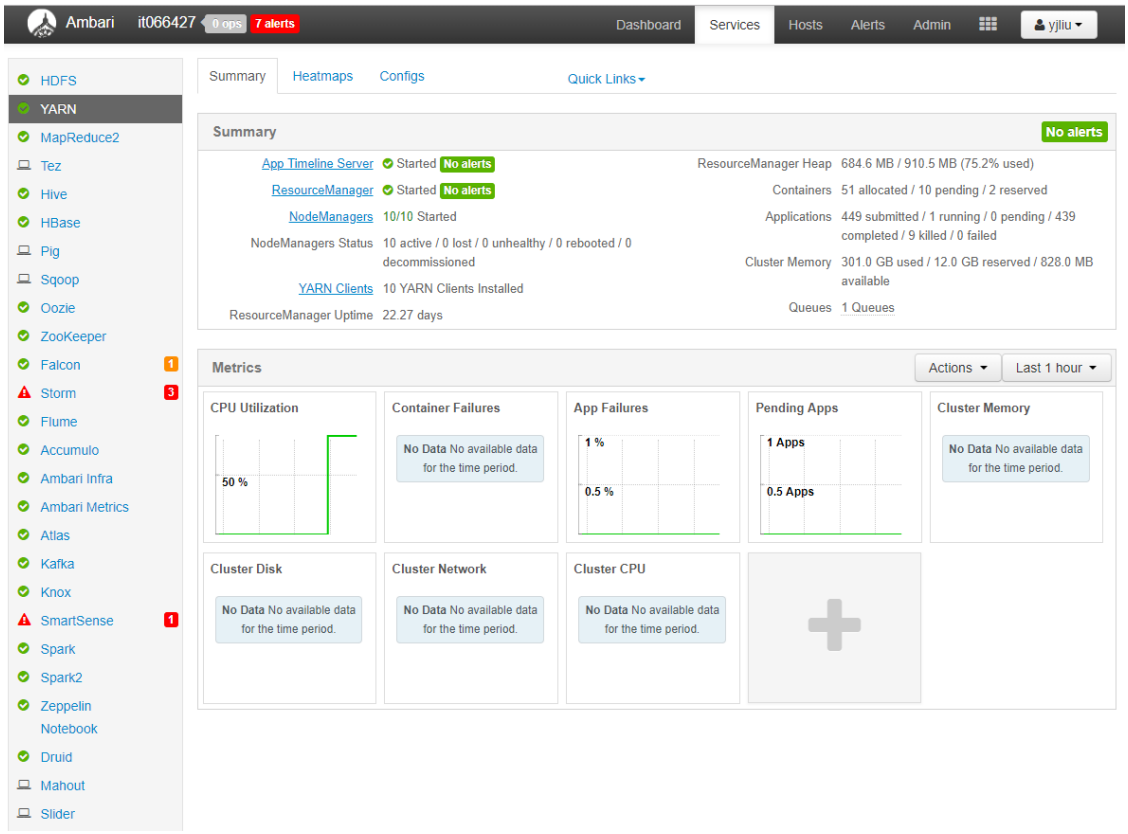

Figure 3.4: Ambari UI

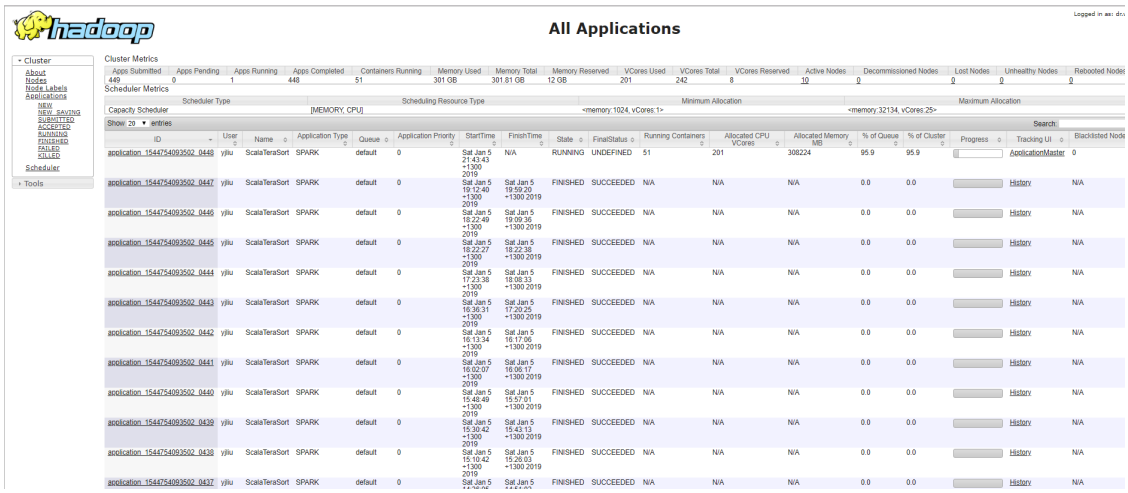

Figure 3.5: YARN UI

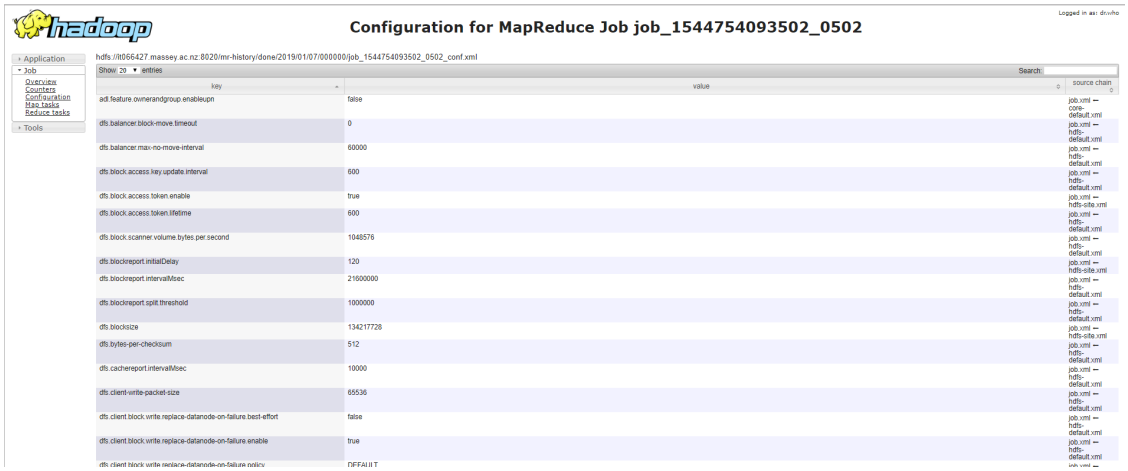

Figure 3.6: Configuration function

The last step is changing the configuration of the chosen parameters. We can change any parameters we want from the command line and use Yarn UI to monitor the situation of the jobs. The Listing 3.8 shows the MapReduce WordCount jobs with customized parameter settings realed to the resource utilization.

```
/usr/hdp/current/hadoop-client/bin/hadoop jar /usr/hdp/current/hadoop-client/../hadoop-
    mapreduce/hadoop-mapreduce-examples.jar wordcount -D mapreduce.reduce.memory.mb=16384 -
    D mapreduce.reduce.cpu.vcores=1 -D mapred.reduce.tasks=25 hdfs://it066427:8020//user/
    yjliu//HiBench/wordcount50g hdfs://it066427:8020//user/yjliu//HiBench/Wordcount/Output
```

Listing 3.8: Wordcount with resource utilization settings

3.5 Data collection

In this section, how to collect the input data are going to be discussed. Since the execution time can present the efficiency of the workloads directly, we decide to use it to present our results. Also, we are going to test each experiment for 5 times to ensure the result accuracy. Then, we can get the average execution time through the five experiments and calculate the average standard deviation based on the results. In addition, we set the threshold for the standard deviation as 20% which means the execution times that shows higher deviate trend will be defined as the outliers. If there is one outlier exists in one experiment, we will drop it and re-calculate the average execution time based on the other four. We hope this data collection method can help us get the results that shows the real performance of our cluster.

Chapter 4

Methodology

For our experiments, we apply three kinds of jobs: aggregation job, shuffle job and iterative job. All our jobs have two implement ways which are Spark and MapReduce. Also, we ran all the jobs on our research cluster which owns one master node and 9 slave nodes. Since each slave node provides 42GB RAM and 8 CPU cores, we are able to assign 420 GB RAM and 104 CPU cores to each of the job. Besides, we selected Yarn as our resource manager which can help us monitor the situation of each working nodes as well as track the details of each job with its history serve. In addition, we select three workloads already provided by Hibench to represent the three types of jobs: Wordcount (aggregation job), TeraSort (shuffle job) and K-means (iterative job). Next, we are going to introduce our first experiment: Wordcount.

4.1 Wordcount

4.1.1 Input Datasets

The input data is produced by *RandomTextWriter* and is divided into 10 pieces from 500GB to 50GB with 50GB as the interval. We choose the one with default settings as the benchmark and use it as comparison group to visualize the result of the 10 different datasets. Also, we divide our datasets as three categories: small, intermediate and large. Each of the category represents one possible scale of the dataset and contains three or four datasets. We hope this subdivision can simulate the real-world situation and help us visualize the results difference in detail.

4.1.2 MapReduce experiment

For our first experiment in MapReduce, we modified the wordcount examples provided by *Hadoopexamples* package. Since all our input data is under *sequencefile* format, we need to modify the source code to let it accept the *sequencefile*. Besides, our first experiments

list three aspects that may affect the cluster performance which are resource utilization, input splits and map-side parameters. Next is about introducing all of them in details.

Resource utilization

First is about something related to resource utilization. Since MapReduce framework only provides map and reduce function to handle the problem, most of the resource is spent on the map tasks and reduce tasks. Thus, we are going to compare the performance difference by tuning the map and reduce resource.

For MapReduce jobs, there are two kinds of resource that we can assign to different jobs which are memory and CPU cores. Thus, we need to set the resource for both map tasks and reduce tasks. For our cluster, the default resource settings for the map tasks are 7GB memory and 1 vcore while the Reduce tasks are 14GB RAM and 1 vcore which means when one Mapper or Reducer launches, it will be given these resources for processing different tasks. Based on that, we decide to change the memory and CPU cores for each map tasks and reduce tasks to see whether there are some difference. For our experiments, we set up three groups of parameters with seven different settings: the default group is 7GB memory with 1 vcore for map tasks, 14GB memory and 1 vcore for reduce tasks. The reducing group includes three settings which are 4GB memory and 1 vcore for map tasks and 8 GB memory and 1 vcore for reduce tasks, 5GB memory and 1 vcore for map tasks and 10 GB memory and 1 vcore for reduce tasks and 6GB memory and 1 vcore for map tasks and 12 GB memory and 1 vcore for reduce tasks. Similarly, the increasing group also includes three settings that are 8GB memory with 2 vcores for map tasks and 16GB memory with 2 vcores for reduce tasks, 9GB memory with 2 vcores for map tasks and 18GB memory with 2 vcores for reduce tasks and 10GB memory and 2 vcores for map tasks and 20GB memory and 2 vcores for reduce tasks. By comparing increasing and reducing the resource for map and reduce tasks, we can get whether resource can affect the job performance.

Input splits

Inputsplits controls the number of map tasks. As we mentioned above, the default block size of MapReduce is 128MB which means all the data in HDFS will be splits into different blocks and save on the different data nodes. Then MapReduce will use input splits to record the starting and end position of the job and match them to different map tasks. Since each map tasks correspond to one input split, tuning input splits can affect the number of map tasks. For our cluster, the default input split is 128MB which means each map tasks will process 128MB data. Thus, we decide to increase that settings forcing MapReduce to process large amount of data for each map tasks. We also set up three groups of parameters: 256MB, 512MB and 1024MB. By setting these parameters on the

command line when we run the MapReduce job, we can control the numbers of map tasks and thus improve the efficiency by reducing the I/O pressure. Through these settings, we want to know whether the number of map tasks or the input splits can affect the job efficiency.

Map-side parameters

Map-side parameters represent some parameters related to I/O process when map tasks merging the data at the end of the map stage. Among all of them, the most important two should be *io.sort.mb* and *io.sort.factor*. *io.sort.mb* refers to the size of the buffer when the map tasks spills while *io.sort.factor* refers to how many spill files can be process together. For our experiments, the default value of *io.sort.mb* is 2GB which means the map tasks will spill as an intermediate files when the output reach to 2GB. Also, the default value of *io.sort.factor* equals 100 which means MapReduce will merge 100 spill files at one time. From our understanding, although the larger buffer size and higher parallelism can reduce I/O pressure, large intermediate files may lead map tasks merge slowly and thus reduce the efficiency. Thus, we choose four groups of parameters which are default settings and the other three settings (1.5GB and 75, 1GB and 50, 0.5GB and 25). By comparing the different settings of the merge process, we can understand whether the I/O parameters can affect the job results.

Execution details

In general, our first experiment needs to utilize each dataset for 13 times. 6 for increasing and reducing map and reduce resource, 3 for input splits and 4 for I/O factors. Next, we are going to explain how to execute the experiments on our own cluster and tune the parameters.

Listing 4.1 illustrates the way we submit our jobs on the cluster related to resource utilization. For MapReduce jobs, we need to locate the jar package and point out the class file the job needs. Then, we need to set some environment parameters for our job. All the parameter settings are implemented by *–D* command. For our experiments, we set the memory and vcores for each map and reduce tasks and change the input format as sequencefile. Finally, we give the HDFS directories as the location of input and output file to make sure the job run properly. Thus, we just need to change the *mapreudce.map.memory.mb* and *mapreudce.map.cpu.vcores* for map tasks and *mapreudce.reduce.memory.mb* and *mapreudce.map.cpu.vcores* for reduce tasks and change the HDFS directory to test the experiments related to resource utilization.

```
1 /usr/hdp/current/hadoop-client/bin/hadoop  jar /usr/hdp/current/hadoop-client/../hadoop-
      mapreduce/hadoop-mapreduce-examples.jar wordcount -D mapreduce.map.memory.mb=7168 -D
      mapreduce.map.cpu.vcores=1 -D mapreduce.reduce.cpu.vcores=1 -D mapreduce.reduce.memory.
      mb=14436 -D mapreduce.inputformat.class=org.apache.hadoop.mapreduce.lib.input.
      SequenceFileInputFormat -D mapreduce.outputformat.class=org.apache.hadoop.mapreduce.lib.
```

```
output.SequenceFileOutputFormat -D mapreduce.job.inputformat.class=org.apache.hadoop.
mapreduce.lib.input.SequenceFileInputFormat -D mapreduce.job.outputformat.class=org.
apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat hdfs://it066427:8020//user/
yjliu//HiBench/wordcount50g/Input hdfs://it066427:8020//user/yjliu//HiBench/Wordcount/
Output
```

Listing 4.1: The MapReduce experiment about the resource utilization for WordCount

Listing 4.2 presents the way we implement the experiments about the input splits. We add two parameters to configure the input splits which are: *mapred.min.split.size* and *mapred.max.split.size*. By forcing both to a specific value, we can control the input splits as a given value.

```
1 /usr/hdp/current/hadoop-client/bin/hadoop jar /usr/hdp/current/hadoop-client/../hadoop-
mapreduce/hadoop-mapreduce-examples.jar wordcount -D mapred.min.split.size=268435456 -D
mapred.max.split.size=268435456 -D mapreduce.inputformat.class=org.apache.hadoop.
mapreduce.lib.input.SequenceFileInputFormat -D mapreduce.outputformat.class=org.apache.
hadoop.mapreduce.lib.output.SequenceFileOutputFormat -D mapreduce.job.inputformat.class=
org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat -D mapreduce.job.
outputformat.class=org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat hdfs:
//it066427:8020//user/yjliu//HiBench/wordcount50g/Input hdfs://it066427:8020//user/
yjliu//HiBench/Wordcount/Output
```

Listing 4.2: The MapReduce experiment about the input splits for WordCount

Listing 4.3 reveals the way we submit the jobs related to Map-side parameters. From the figure, we can see that we add two parameters: *mapreduce.task.io.sort.mb* and *mapreduce.task.io.sort.factor*. By tuning these two, we can find the relationship between jobs and I/O pressure.

```
1 /usr/hdp/current/hadoop-client/bin/hadoop jar /usr/hdp/current/hadoop-client/../hadoop-
mapreduce/hadoop-mapreduce-examples.jar wordcount -D mapreduce.task.io.sort.mb=1024 -D
mapreduce.task.io.sort.factor=50 -D mapreduce.inputformat.class=org.apache.hadoop.
mapreduce.lib.input.SequenceFileInputFormat -D mapreduce.outputformat.class=org.apache.
hadoop.mapreduce.lib.output.SequenceFileOutputFormat -D mapreduce.job.inputformat.class=
org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat -D mapreduce.job.
outputformat.class=org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat hdfs:
//it066427:8020//user/yjliu//HiBench/wordcount50g/Input hdfs://it066427:8020//user/
yjliu//HiBench/Wordcount/Output
```

Listing 4.3: The MapReduce experiment about the Map-side parameters for WordCount

4.1.3 Spark experiment

For our first experiments in Spark, we also modify the source code from *Sparkexample package* and make them possible to accept the sequencefile as the input data. In addition, there are many aspects from our research that may affect the efficiency of the Spark jobs: resource utilization, input splits and parallelism. Next, we are going to introduce each of them in details.

Resource utilization

Different from MapReduce framework, Spark proposes its own strategy to assign resource to different executors. The executors will be launched under different work nodes and each of them will charge many tasks to implement parallel computing. Also, Spark allows the users to set the number of executors as well as memory and vcores for each executor. For our cluster, the default setting is 2 executors with 2GB memory and 1 vcore. Apparently, this setting can't fully utilize the resource of our cluster and thus we decide to use our own settings.

Based on Spark user manual, the proper setting of number of executors ranges from 50 to 100. Thus, we decide to set 6 groups of parameters to test the efficiency of Wordcount which are 50 executors with 8GB memory and 4 vcores, 60 executors with 7GB memory and 4vcores, 70 executors with 6GB memory and 3 vcores, 80 executors with 5GB memory and 3 vcores, 90 executors with 4GB memory and 2 vcores and 100 executors with 4GB memory and 2 vcores. By conducting this experiment, we hope to see whether the different resource strategy affect the result.

Input Splits

Similar to MapReduce, Spark also reads the input data from HDFS and thus the input splits can affect the amount of map tasks as well. As the comparison of the MapReduce input splits experiment, we also set the input splits as 256MB, 512MB and 1024MB. By forcing each map tasks to process larger scale of data, we want to find out whether I/O pressure is the key factor for both MapReduce and Spark. Also, we want to see how much I/O pressure can affect the efficiency of both jobs.

Parallelism

Parallelism in Spark is utilized to describe how many tasks are going to run together for each Spark stage. After finished resource allocation, we need to set parallelism to ensure all the executors are fully occupied. According to the Spark manual, parallelism should be large enough to make sure that all the resources are fully used. For our cluster, the default parallelism is not set which means some resource will be wasted when several tasks end early. To improve the efficiency, we set five groups of parallelism to see whether these can improve the efficiency.

Since the largest dataset of our experiment is 500GB which equals to 4000 map tasks and our default number of executors is 50, the parallelism should be 4000 divide 50 and the result is 80. Thus, we need to set the parallelism bigger than 80 to make sure once one tasks finishes early, the resource will be assigned to other tasks. Based on Spark manual, we choose five parameters from 100 to 500 as the parallelism to see whether enlarge the parallelism multiple times can affect Spark efficiency.

Execution details

Thus, our experiment in Spark needs to utilize each dataset for 15 times, 6 times for resource utilization, 4 times for input splits and 5 times for parallelism. Then, the details of how we implement our jobs are shown below.

Listing 4.4 illustrates how we submit our jobs on the cluster through *spark-submit* command. For Spark jobs, we need to set the class file through *–class* command and set the master model through *–master* command. Then, we need to assign different resource for each executor. For our experiments, we assign different memory and vcores for each executor to see the difference. Finally, we need to provide the HDFS location as input and output directories.

```
1 /usr/hdp/current/spark2-client/bin/spark-submit --class com.intel.hibench.sparkbench.micro.
    ScalaWordCount --master yarn-client --num-executors 50 --executor-cores 4 --executor-
    memory 8g /home/yjliu/Hibench/HiBench-master/sparkbench/assembly/target/sparkbench-
    assembly-7.1-SNAPSHOT-dist.jar hdfs://it066427:8020//user/yjliu//HiBench/wordcount50g/
    Input hdfs://it066427:8020//user/yjliu//HiBench/Wordcount/Output
```

Listing 4.4: The Spark experiment about the resource utilization for WordCount

Listing 4.5 presents how we submit Spark jobs related to input splits. Since Spark shares the parameters with MapReduce, we need to tune this parameter by adding the two variables: *spark.hadoop.mapreduce.input.fileinputformat.split.minsize* and *spark.hadoop.mapreduce.input.fileinputformat.split.maxsize*. Then, we need to configure these two parameters by *–conf* command and they allow us to change any values we want.

```
1 /usr/hdp/current/spark2-client/bin/spark-submit --class com.intel.hibench.sparkbench.micro.
    ScalaWordCount --master yarn-client --num-executors 50 --executor-cores 4 --executor-
    memory 8g --conf spark.hadoop.mapreduce.input.fileinputformat.split.maxsize=268435456
    --conf spark.hadoop.mapreduce.input.fileinputformat.split.minsize=268435456 /home/
    yjliu/Hibench/HiBench-master/sparkbench/assembly/target/sparkbench-assembly-7.1-SNAPSHOT
    -dist.jar hdfs://it066427:8020//user/yjliu//HiBench/wordcount50g/Input hdfs://it066427
    :8020//user/yjliu//HiBench/Wordcount/Output
```

Listing 4.5: The Spark experiment about the input splits for WordCount

Listing 4.6 shows how we execute the Spark jobs related to parallelism. For our experiments, we add the parameter *spark.default.parallelism* and implement it by *–conf* command. Then, we just need to change the value to see the difference.

```
1 /usr/hdp/current/spark2-client/bin/spark-submit --class com.intel.hibench.sparkbench.micro.
    ScalaWordCount --master yarn-client --num-executors 50 --executor-cores 4 --executor-
    memory 8g --conf spark.default.parallelism=100 /home/yjliu/Hibench/HiBench-master/
    sparkbench/assembly/target/sparkbench-assembly-7.1-SNAPSHOT-dist.jar hdfs://it066427
    :8020//user/yjliu//HiBench/wordcount50g/Input hdfs://it066427:8020//user/yjliu//HiBench/
    Wordcount/Output
```

Listing 4.6: The Spark experiment about the parallelism for WordCount

4.2 K-means

Next, we describe our second experiment: K-means. As an iterative task, it needs to save the previous results on somewhere for further processing. MapReduce and Spark provide different strategies to handle the problems. MapReduce framework provides the function to save the intermediate data on HDFS while Spark allows the users to save the data on memory or disk. For our experiments, we decide to test both strategies to see what is the difference between the two. Our plan is as follows.

4.2.1 Input Datasets

The input data are created by *GenKMeansDataset*[1] function provided by HiBench suite. The datasets include 6 parts ranging from 50GB to 150GB with 25GB as the interval. Since our cluster only owns 420GB memory and 250 vcores, we decide to use 300 GB memory and 210 vcores for cache operation. Besides, since 20% of the memory for each executor will be used for shuffle process and each executor needs to give some space for *memoryOverHead* which stands for the space for JVM as well as store some important files. Thus, the upper line of the size of our dataset is set as 150GB to avoid data overflow. Also, the datasets are divided into two groups: small and intermediate. These groups can help us visualize the difference as well as presents the improvement in detail. We hope these datasets can simulate the real situation when processing iterative tasks for MapReduce and Spark.

4.2.2 MapReduce experiment

We implement our MapReduce job with Mahout K-means function. Since the input data is under *sequencefile* format, we can use the data directly to get the result. Although MapReduce job is not good at processing the iterative jobs because of the repeated I/O operation, we keep it there to see the difference between hard drive reading and memory storage. Thus, we use MapReduce job results as the benchmark to compare with the Spark jobs with different persist strategy. For our experiments, we are going to run all the experiments with the default settings and hope to get the obvious result with the help of our datasets.

Execution details

As we explained above, our MapReduce experiment is simple. It just needs to execute all the datasets for once with the default settings. The details are as follows.

Listing 4.7 illustrates the implementation of K-means in MapReduce. From the listing, we can see that there are several parameters need to be set to make sure the jobs

[1]The function to create k-means data.

can be executed properly. The first two are input and output directories and they are implemented by *–i* and *–o* command. The other parameters are all related to K-means algorithm: the number of K (*-K*), the number of iteration (*-x*), the convergence Delta (*-cd*), the distance measurement (*-dm*) and the execution model (*-xm*). We can test the performance by tuning the parameters provided above.

```
mahout kmeans -i hdfs://it066427:8020//user/yjliu/HiBench/Kmeans2/Input/samples -c hdfs://
    it066427:8020//user/yjliu/HiBench/Kmeans2/Input/cluster -o hdfs://it066427:8020//user/
    yjliu/outputyyy -x 10 -cd 0.5 -dm org.apache.mahout.common.distance.
    EuclideanDistanceMeasure -xm mapreduce
```

Listing 4.7: The MapReduce experiment about K-means

4.2.3 Spark experiment

For K-means in Spark, the main effort of our second experiment is gathering there. There are three persistence levels provided by Spark which offers three ways to save the intermediate results: MEMORY_ONLY, MEMORY_AND_DISK and DISK_ONLY. Next, we are going to introduce each of them in details.

MEMORY_ONLY

As the most common persist strategy for Spark cache operation, MEMORY_ONLY provides the function to saves all the intermediate data on the RAM and thus improve the efficiency of the jobs dramatically. Also, it is the best strategy for Spark jobs without considering the resource utilization. For our experiments, we set 30 as executor numbers, 8GB as executor memory and 3 as executor vcrores to fully utilize our cluster resource. Also, we set the *memoryOverHead* as 2GB for each executor. Then, we run all the datasets with MEMORY_ONLY strategy and set them as the Spark benchmark. These benchmarks can help us comparing the difference between different Spark persist strategy and MapReduce jobs later.

MEMORY_AND_DISK

As a compromised persist strategy, MEMORY_AND_DISK provides the function to save the data on the disk and memory together. The strategy is like that: Spark utilizes the unserialized Java objects as the storage format and give priority to save data in memory. If there is not enough memory to store all the data, Spark will write the others on the disks. This strategy is suitable for the situation that the RDDs are too huge to save on the memory. Although it may reduce the efficiency, it can guarantee the jobs run smoothly and thus accepted by many users.

For our experiment, we set 30 executors with 8GB memory, 2GB *memoryOverHead* and 7 vcores to fully utilize the resource. Thus, we have 300GB memory and 210 vcores

totally. Then, we set the threshold of memory utilization as 50% which means our experiments will save 90% to 50% of the data on memory with 10% as the interval. The reason is because memory should be the main resource for Spark to allocate and if there are more data saved on disks than memory, the efficiency may be lower than save all of them on disks. Also, since we have 5 datasets from 50GB to 150GB, we need to calculate how much data needs to save on memory and disks for each. Since *spark.storage.memoryFraction* can control the percent of each executor for caching operation, we needs to assign each of them with a value from 90% to 50%. Take 150GB as an example: since the threshold is 50%, our experiments need to save 75GB (50%), 90GB (60%), 105GB (70%), 120GB (80%) and 135GB (90%) data into memory. Based on that we can get that the *memoryFraction* values are 0.36, 0.43, 0.5, 0.57, 0.64. We hope this experiment can helps us understand how much the disks storage strategy can affect the results.

DISK_ONLY

As the most rarely used persist strategy, DISK_ONLY provides the function to save all the intermediate data on the disks and thus reduce the efficiency of Spark jobs. Similar to MapReduce framework, Spark also provides the storage strategy and can save the memory resource to the greatest extent. For our experiments, we also set 30 as executor numbers, 8GB as executor memory, 2GB as *memoryOverHead* and 7 as executor vcrores to fully utilize our cluster resource. Then, all the datasets will be run under this storage strategy to see the similarities and differences of the MapReduce jobs.

Execution details

In total, our Spark experiments requires to run all the datasets for 7 times, 1 time for MEMORY_ONLY, 5 times for MEMORY_AND_DISK and 1 time for DISK_ONLY. Since each spark job can utilize one cache strategy, we need to modify the source codes and compress them as three different jar packages to execute on our cluster. The details of our experiment are shown below.

Listing 4.8 illustrates how Spark jobs work on the cluster with DISK_ONLY strategy. From the listing, we can find that the class file, master model and resource parameter need to be set which are similar to that on the last Spark experiment. In addition, we also set one parameter called *spark.yarn.executor.memoryOverhead* which represent the space to run the JVM as well as store some important files. Also, we need to configure the directory of jar files and give the k value and number of iterations to make the jobs work.

```
1 spark-submit --class com.intel.hibench.sparkbench.ml.DenseKMeans --master yarn-client --num-
      executors 30 --executor-cores 8  --executor-memory 8g --driver-memory 2g --conf spark.
      yarn.executpr.memoryOverhead=2048  /home/yjliu/Hibench/HiBench-master/sparkbench/
      assembly/target/sparkbench-assembly-7.1-SNAPSHOT-distDiskOnly.jar -k 5 --numIterations 5
        hdfs://it066427:8020//user/yjliu//HiBench/Kmeans2/Input/samples
```

Listing 4.8: The Spark experiment about K-means DISK_ONLY

Listing 4.9 presents how Spark jobs work with MEMORY_ONLY strategy. For our experiments, we enable one function called *useLegacyMode* to allow the users to assign the memory for cache operation. Then we set the parameter called *spark.storage.memoryFraction* to change the percentage of executor memory used for cache operation. Finally, we change the jar package and set the k value and number of iterations to make the job works.

```
spark-submit --class com.intel.hibench.sparkbench.ml.DenseKMeans --master yarn-client --num-executors 30 --executor-cores 8 --executor-memory 8g --driver-memory 2g --conf spark.yarn.executpr.memoryOverhead=2048 --conf spark.memory.useLegacyMode=true --conf spark.storage.memoryFraction=1 /home/yjliu/Hibench/HiBench-master/sparkbench/assembly/target/sparkbench-assembly-7.1-SNAPSHOT-distmemoryonly.jar -k 5 --numIterations 5 hdfs://it066427:8020//user/yjliu//HiBench/Kmeans2/Input/samples
```

Listing 4.9: The Spark experiment about K-means MEMORY_ONLY

Listing 4.10 shows how Spark executes the experiment of MEMORY_AND_DISK. All the setting are similar to the MEMORY_ONLY experiment, we just need to change the jar package as well as the value of *spark.storage.memoryFraction* to see the difference.

```
spark-submit --class com.intel.hibench.sparkbench.ml.DenseKMeans --master yarn-client --num-executors 30 --executor-cores 8 --executor-memory 8g --driver-memory 2g --conf spark.yarn.executpr.memoryOverhead=2048 --conf spark.memory.useLegacyMode=true --conf spark.storage.memoryFraction=0.5 /home/yjliu/Hibench/HiBench-master/sparkbench/assembly/target/sparkbench-assembly-7.1-SNAPSHOT-distMemoryandDIsk.jar -k 5 --numIterations 5 hdfs://it066427:8020//user/yjliu//HiBench/Kmeans2/Input/samples
```

Listing 4.10: The Spark experiment about K-means MEMORY_AND_DISK

4.3 TeraSort

The third experiment is Terasort. As a kind of shuffle job, Terasort spends the most resources and execution time on transferring data from map side to reduce side. Since shuffle performance can be recognized as the key point of the MapReduce and Spark jobs, Terasort can help us understand how much the parameters settings can affect the results. For our experiments, we take three kinds of parameters into consideration: resource utilization, input splits and reduce-side parameters. The plan of our experiments is shown below.

4.3.1 Input Datasets

The input data is created by *TeraGen* function provided by *Hadoop-example package*. Our datasets are divided into 10 pieces ranging from 50GB to 500GB with 50GB as interval and are classified as three categories based on its sizes. The small category includes 50GB,

100GB and 150GB, the intermediate one includes 200GB, 250GB and 300GB while the big one includes 350GB, 400GB, 450GB and 500GB. Since our datasets are various enough to cover different scale of jobs, we hope these can help us simulate the real-world situation of shuffle jobs and propose the insights to improve the job efficiency.

4.3.2 MapReduce experiment

For our third experiment, we utilize the function directly from *Hadoop-example package* to implement our design. Also, we modify the source code a little bit to let it accept the sequencefile as the input data. During the experiment, the most important part is changing the parameters from three aspects and compare the performance of them. Thus, we are going to talk about these three aspects and visualize the results later. The plan is below.

Resource utilization

The first aspect is about assigning different resource for our jobs. There are two kinds of resources that we can assign to different map and reduce tasks: memory and vcores. Thus, we are going to design our experiments by giving different memory and vcores to map and reduce tasks.

For our experiments, the default resource settings for map tasks is 7GB memory with 1 vcore and the reduce tasks is 14GB memory with 1 vcore. Since Terasort belongs to shuffle jobs and our cluster owns 420GB memory and 250 vcores, we decide to set the map tasks as the default value while changing the parameters related to reduce tasks. We also put number of reduce tasks into consideration because the default settings for reduce tasks is only one which may strongly affect our job efficiency. To fully utilize our resource, we set the reduce memory from 10GB to 18GB with 2GB as interval. Then, we assign different reduce numbers to different settings: 40 reduce tasks are launched and each of them are given 10GB memory, 35 reduce tasks are given 12GB memory for each, 30 reduce tasks are given 14GB memory for each, 25 reduce tasks are given 16GB memory and 20 reduce tasks are given 20GB memory for each. Besides, we provide different vcores for different memory settings: we give 1 vcore for reduce tasks for the memory between 10GB and 14GB while giving 2 vcores for memory larger than 14GB. From tuning these parameters, we want to know whether more reduce tasks with small resource or less reduce tasks with more resource can improve the efficiency.

Input Splits

The second aspect is input splits which stands for how many map tasks our jobs need to process. The default setting of our MapReduce is 128MB which means each map tasks contains 128MB data. Since the shuffle efficiency depends on both the size of shuffle jobs

as well as the number of shuffle jobs, we decide to take the input splits into consideration to see which strategy can improve the efficiency. Similar to the first experiment, we set the input splits as 256MB, 512MB and 1024MB. From the experiments, we not only want to see whether the shuffle performance can be affected by the number or the size of map tasks, but also curious about if the result of the aggregation job is similar to the shuffle jobs.

Reduce-side parameters

The third aspect is about the parameters related to reduce tasks. Although the shuffle process is not equal to the reduce process, MapReduce integrates both of them under reduce parameters. Thus, these parameters can not only affect shuffle performance, but also control the efficiency of reduce tasks. Among all of them, we select two that may strongly affect the shuffle performance: *MapReduce.reduce.shuffle.parallelcopies* and *MapReudce.task.io.sort.factor*. The first one represents the number of threads that copy map output data and the second on represents the number of files that combine together. These two parameters control the parallelism and the size of each shuffle tasks and thus need to be considered. For our experiments, the default value of *parallelcopies* is 100 while the *io.sort.factor* is 30. To ensure considering both aspects, we decide to set four groups of the values that contains both of the increasing and decreasing situation and compare them to the default setting. For our experiments, we set 200, 150, 75 and 50 as *parallelcopies* while 60, 45, 20 and 15 as the *io.sort.factor*. Through the experiments, we hope to understand how much these reduce-side parameters can affect the MapReduce job efficiency.

Execution details

Totally, we need to execute all the datasets for 12 times, 5 times for resource utilization, 3 times for input splits and 4 times for reduce-related parameters. The details of how we implement the Spark jobs are shown below.

Listing 4.11 illustrates how resource utilization experiments work for Terasort. For the MapReduce jobs, we need to set the directory of jar package and the class file we used. Then, we need to set the resource utilization parameters with *–D* function. Thus, we add two parameters to control the shuffle process: *mapreduce.reduce.memory.mb* and *mapred.reduce.tasks*. By tuning these two parameters, we can find out whether more resource for reduce tasks can affect the efficiency of shuffle jobs.

```
1 /usr/hdp/current/hadoop-client/bin/hadoop jar /usr/hdp/current/hadoop-client/../hadoop-
    mapreduce/hadoop-mapreduce-examples.jar terasort -D mapreduce.reduce.memory=16384 -D
    mapreduce.reduce.cpu.vcores=1 -D mapred.reduce.task=25 hdfs://it066427:8020//user/yjliu
    //HiBench/Terasort/Terasort10g hdfs://it066427:8020//user/yjliu//HiBench/Terasort/Output
```

Listing 4.11: The MapReduce experiment about resource utilization for Terasort

Listing 4.12 presents how MapReduce jobs works with different input splits. For the experiments, we also add *mapred.min.split.size* and *mapred.max.split.size* and force them to the same value to see the difference.

```
1 /usr/hdp/current/hadoop-client/bin/hadoop jar /usr/hdp/current/hadoop-client/../hadoop-
      mapreduce/hadoop-mapreduce-examples.jar terasort -D mapreduce.reduce.memory=16384 -D
      mapreduce.reduce.cpu.vcores=1 -D mapred.reduce.task=25 -D mapred.min.split.size
      =268435456 -D mapred.max.split.size=268435456 hdfs://it066427:8020//user/yjliu//
      HiBench/Terasort/Terasort10g hdfs://it066427:8020//user/yjliu//HiBench/Terasort/Output
```

Listing 4.12: The MapReduce experiment about input splits for Terasort

Listing 4.13 shows how the experiments related to reduce-side parameters work. From the figure, we can find that two new parameters are available which are *mapreduce.reduce .shuffle.parallelcopies* as well as *mapreduce.task.io.sort.factor*. These two parameters can help us find the difference by tuning the reduce-related parameters.

```
1 /usr/hdp/current/hadoop-client/bin/hadoop jar /usr/hdp/current/hadoop-client/../hadoop-
      mapreduce/hadoop-mapreduce-examples.jar terasort -D mapreduce.reduce.memory=16384 -D
      mapreduce.reduce.cpu.vcores=1 -D mapred.reduce.task=25 -D mapreduce.reduce.shuffle.
      parallelcopies=15 -D mapreduce.task.io.sort.factor=50 hdfs://it066427:8020//user/yjliu
      //HiBench/Terasort/Terasort10g hdfs://it066427:8020//user/yjliu//HiBench/Terasort/
      Output
```

Listing 4.13: The MapReduce experiment about reduce-related parameters for Terasort

4.3.3 Spark experiment

Regarding to the experiments in Spark, we utilize the function provided by Hibench and compress them into the jar package to execute it on our own cluster. Besides, we also set some related parameters that may improve the efficiency of Spark jobs. Similar to MapReduce experiments, there are three kinds of parameters which are the number of reduce tasks, input splits and shuffle-related parameters. The details of our experiment are discussed below.

Resource utilization

The first aspect is the number of reduce tasks. Since Spark adopts different ways to assign resources compared to MapReduce, we need to give different executors memory and vcores before running our jobs. Next is about setting the number of partitions for Terasort. Because the number represents how much data in each partition and is correspond to the number of reduce tasks, we take it into consideration and put it at the first place.

For our experiments, we set 70 executors and give each of them 6GB memory as well as 3 vcores. These settings reach the limit of our cluster and thus can help us fully utilize our resource. Then, the amount of reduce tasks is set from 20 to 40 with 5 as the interval. We keep this setting same to the MapReduce jobs to make sure both of MapReduce and Spark meet the same situation. Also, we change the replication factor as 1 which already implemented by MapReduce to improve the job efficiency. Through the experiments, we want to see the difference for the shuffle jobs between MapReduce and Spark based on the same settings.

Input splits

The second aspect is input splits. Similar to MapReduce experiments above, they are about changing the sizes of each map tasks and thus reducing the shuffle pressure. For our experiments, the input splits are set as 256MB, 512MB and 1024MB. By enlarging the values multiple times, we hope to see whether the different sizes of map tasks can strongly affect the efficiency of shuffle jobs. Besides, we can compare the performance between MapReduce and Spark by implementing both with same settings.

shuffle-related parameters

The last aspect is about parameters related to the shuffle process. Based on our study, there are two parameters that may affect the shuffle process which are *spark.shuffle.file.buffer* and *spark.reducer.maxSizeInFlight*. The first one stands for the size of in-memory buffer for each shuffle output stream. This buffer can improve the shuffle efficiency by reducing the number of system calls as well as disk seeks when creates the intermediate shuffle files. The second one represents the buffer size of the shuffle read tasks which decides how much data can be fetched for once. Also, enlarging this value can reduce the fetch times which may affect the times of network transformation times to improve the efficiency.

For our experiments, these two parameters are not included in our Spark jobs. Thus, we select four groups of values to test whether they can affect the efficiency. These values are 16K and 32MB, 32K and 48MB, 64K and 96MB and 128K and 192MB. By enlarging both values multiple times, we want to see whether these two parameters can dramatically improve the shuffle efficiency. Also, these parameters provide us a clue that how much the shuffle-related parameters can improve the job efficiency compared to MapReduce reduce-related parameters.

Execution details

Totally, our experiment needs to run all our datasets from 50GB to 500GB for 12 times, 5 times for the number of reduce tasks, 3 times for input splits and 4 times for shuffle-related parameters. The results of the experiments can be seen below.

Listing 4.14 and 4.15 illustrate how the experiment related to the number of reduce tasks works. Listing 4.14 shows us the basic configuration for Spark jobs and the only thing we need to care is about the replication number. Since MapReduce Terasort set the replication number as 1 to improve the efficiency, we also set it as 1 for Spark to reduce the execution time for copy the results to HDFS. Besides, Listing 4.15 presents how we change the number of the reduce tasks for Spark. By changing the *hibench.default.shuffle.parallelism* from the HiBench conf folder, we can set the number of reduce tasks to compare with MapReduce jobs.

```
/usr/hdp/current/spark2-client/bin/spark-submit --class com.intel.hibench.sparkbench.micro.
    ScalaTeraSort --master yarn-client --num-executors 60 --executor-cores 4 --executor-
    memory 5g --conf spark.hadoop.dfs.replication=1 /home/yjliu/Hibench/HiBench-master/
    sparkbench/assembly/target/sparkbench-assembly-7.1-SNAPSHOT-dist.jar hdfs://it066427
    :8020//user/yjliu//HiBench/Terasort/Terasort10g hdfs://it066427:8020//user/yjliu//
    HiBench/Terasort/Output
```

Listing 4.14: The Spark experiment about number of reduce tasks for Terasort

```
Data scale profile. Available value is tiny, small, large, huge, gigantic and bigdata.
# The definition of these profiles can be found in the workload conf file i.e. conf/
    workloads/micro/wordcount.conf

hibench.scale.profile                   tiny
# Mapper number in hadoop, partition number in Spark
hibench.default.map.parallelism         10

# Reducer nubmer in hadoop, shuffle partition number in Spark
hibench.default.shuffle.parallelism     20
```

Listing 4.15: Changing the reduce number for Spark Terasort

Listing 4.16 shows us how to implement the Spark jobs related to input splits. Same to the previous settings above, we add *mapred.min.split.size* and *mapred.max.split.size* to change the settings of input splits.

```
/usr/hdp/current/spark2-client/bin/spark-submit --class com.intel.hibench.sparkbench.micro.
    ScalaTeraSort --master yarn-client --num-executors 60 --executor-cores 4 --executor-
    memory 5g --conf spark.hadoop.dfs.replication=1 --conf spark.hadoop.mapreduce.input.
    fileinputformat.split.maxsize=268435456 --conf spark.hadoop.mapreduce.input.
    fileinputformat.split.minsize=268435456 /home/yjliu/Hibench/HiBench-master/sparkbench/
    assembly/target/sparkbench-assembly-7.1-SNAPSHOT-dist.jar hdfs://it066427:8020//user/
    yjliu//HiBench/Terasort/Terasort10g hdfs://it066427:8020//user/yjliu//HiBench/Terasort/
    Output
```

Listing 4.16: The Spark experiment about input splits for Terasort

Listing 4.17 reveals the Spark experiments related to shuffle-side parameters. From the listing, we set two new parameters to optimize the shuffle process which are *spark.shuffle.file .buffer* and *spark.reducer,maxSizeInFlight*. By tuning these two, we hope to see whether the shuffle parameters can strongly affect the Spark jobs performance.

```
/usr/hdp/current/spark2-client/bin/spark-submit --class com.intel.hibench.sparkbench.micro.
    ScalaTeraSort --master yarn-client --num-executors 60 --executor-cores 4 --executor-
    memory 5g --conf spark.hadoop.dfs.replication=1 --conf spark.shuffle.file.buffer=16k --
    conf spark.reducer.maxSizeInFlight=48m /home/yjliu/Hibench/HiBench-master/sparkbench/
```

```
assembly/target/sparkbench-assembly-7.1-SNAPSHOT-dist.jar hdfs://it066427:8020//user/
yjliu//HiBench/Terasort/Terasort10g hdfs://it066427:8020//user/yjliu//HiBench/Terasort/
Output
```

Listing 4.17: The Spark experiment about shuffle-related parameters for Terasort

Chapter 5

Experiment Results and Analysis

In this section, the methods to analysis our findings and the results of our experiments are going to be discussed.

5.1 Performance evaluation

This section will explain how to evaluate the performance of the MapReduce and Spark jobs with the three existing experiments. For both MapReduce and Spark jobs, we utilize the execution time to present the efficiency because it can show the difference and efficiency directly. Also, we apply three kinds of graphs to visualize the results of our findings which are execution time graph, gap/relationship graph and increment graph. The first one is the basic application of our experiment results. By showing the execution time of different jobs directly, we can get the basic information of our experiments and compare the difference roughly. To ensure the accuracy of our experiments, we put standard deviation into this graph to present the dispersion of our experiments results. This graph is widely used for all our experiments. The second one is gap/relationship graph which represents the comparison between the benchmark and jobs with different settings. The relationship graph is using the benchmark execution time to divide the others, if it is above 1, the relationship is positive, else it is negative. The gap graph is based on the relationship table and used to express the specific improvement or decline percentage for the tasks compared to the benchmark. These two tables are widely used for experiment one and three. For increment graph, it is about the increment rates when the settings change and it is calculated by using execution time of the new settings to divide the last execution time with previous settings. This graph is mainly used for experiment two. By plotting these graphs, we hope to visualize the experiment results more clearly to others.

5.2 WordCount results

The average execution time of Wordcount for MapReduce and Spark can be shown from Table 5.1 and 5.2. Based on that, we plot the results as different graphs for further analysis. Also, we highlight the best and worst execution time with red and green colours based on different experiment settings. The raw data that includes all the experiments records can be found in the appendix.

Hadoop	50G	100G	150G	200G	250G	300G	350G	400G	450G	500G
Resource utilization										
7G 1vcore:map 14G 1vcore:reduce (default)	212	407	600	790	979	1170	1364	1549	1769	1934
4G 1vcore: map 8G 1vcore:reduce	192	369	538	716	887	1054	1226	1400	1587	1769
5G 1vcore:map 10G 1vcore:reduce	197	378	559	717	898	1065	1242	1445	1625	1805
6G 1vcore: map 12G 1vcore:reduce	201	392	581	755	936	1114	1307	1505	1665	1865
8G 2vcores: map 16G 2vcores:reduce	235	444	653	867	1078	1464	1507	1707	1907	2115
9G 2vcores: map 18G 2vcores:reduce	256	506	737	981	1231	1460	1701	1940	2192	2417
10G 2vcores: map 20G 2vcores:reduce	259	507	744	983	1225	1470	1700	1933	2186	2418
Input Splits:										
128M(default)	212	407	600	790	979	1170	1364	1549	1769	1934
256M	205	364	541	712	896	1047	1223	1380	1537	1720
512M	168	360	531	690	824	1006	1140	1312	1448	1631
1024M	221	370	525	688	841	1022	1132	1279	1429	1572
Map-Side parameters										
default(2047,100)	212	407	600	790	979	1170	1364	1549	1769	1934
i/o.sort.mb=1024 i/o.sort.factor=50	202	394	574	753	946	1130	1314	1486	1674	1853
(1536,75)	206	400	592	771	963	1155	1326	1531	1713	1891
(512,25)	200	372	560	735	922	1091	1266	1461	1608	1769

Table 5.1: The average execution time for MapReduce WordCount

Spark	50G	100G	150G	200G	250G	300G	350G	400G	450G	500G
Resource utilization										
50 executors 8G memory 4 vcores	56	83	120	145	189	201	243	266	285	333
60 7 4	56	94	120	166	200	220	276	297	316	371
70 6 3(default)	58	93	114	169	213	254	286	327	355	410
80 5 3	61	97	127	170	209	249	272	318	328	379
90 4 2	63	95	127	162	184	218	251	290	300	353
100 4 2	60	97	132	161	193	216	247	287	317	341
Input splits										
128M(default)	58	93	114	169	213	254	286	327	355	410
256M	70	119	146	202	260	310	341	365	404	454
512M	78	115	187	216	291	305	340	353	383	464
1024M	90	160	203	249	277	327	355	363	399	480
Parallelism										
100	54	79	113	134	176	199	230	277	287	309
200	60	84	108	151	183	201	234	265	286	328
300	60	93	108	140	189	195	239	275	295	347
400	62	88	114	146	169	207	248	284	292	326
500	60	86	119	144	173	205	240	278	286	342

Table 5.2: The average execution time for Spark Wordcount

5.2.1 Resource utilization

Firstly, we are talking about the resource utilization for MapReduce and Spark. Figure 5.1 shows the execution time of different datasets for MapReduce and Spark. These results are based on the default settings and we recognize them as the benchmark for Spark and MapReduce. From the figure we can see that Spark curve is more sharp which means Spark shows higher efficiency compared to MapReduce. Also, the difference becomes larger with the data size grows that gives us an idea that Spark is better option to process large datasets. Besides, the standard deviation bar for MapReduce is much longer than that on Spark which shows stronger fluctuation when conducting the experiments.

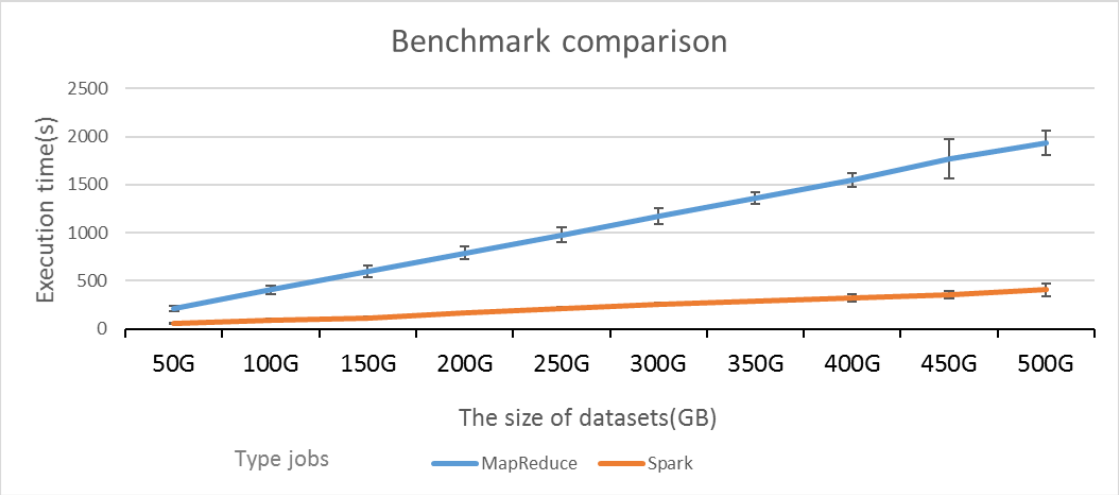

Figure 5.1: The original difference between MapReduce and Spark

Next, Figure 5.2 and Figure 5.3 present the relationship and improvements rates when adopts different resource settings for MapReduce jobs. From the first figure, we can find that: compared to the benchmark, the smaller resource assigned to map and reduce tasks, the higher efficiency MapReduce jobs can be. Also, we can find that the trends for MapReduce jobs which adopts fewer resource than the default settings are stable. It presents the strategy with few resources is suitable for all the datasets we used and can improve the efficiency obviously. On the contrary, the trends for the jobs with more resource than default settings are in chaos. The biggest two lines on the bottom are even overlap and the efficiency is low compared to the benchmark. Then, from the Figure 5.3, we can find how much the different settings affect the efficiency. This figure visualizes the trends in detail and help us understand the positive and negative relation clearly.

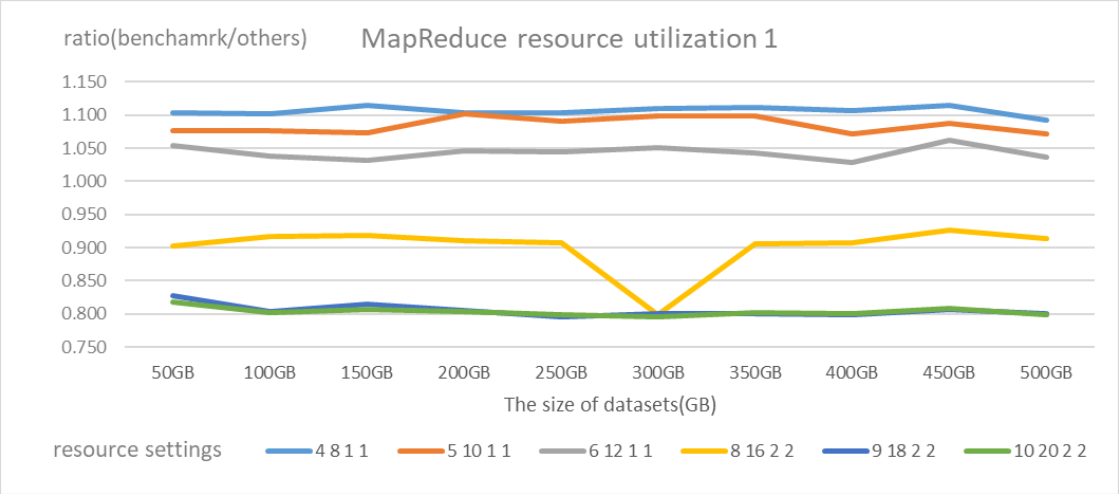

Figure 5.2: The relationship graph for MapReduce after applying different resource strategies

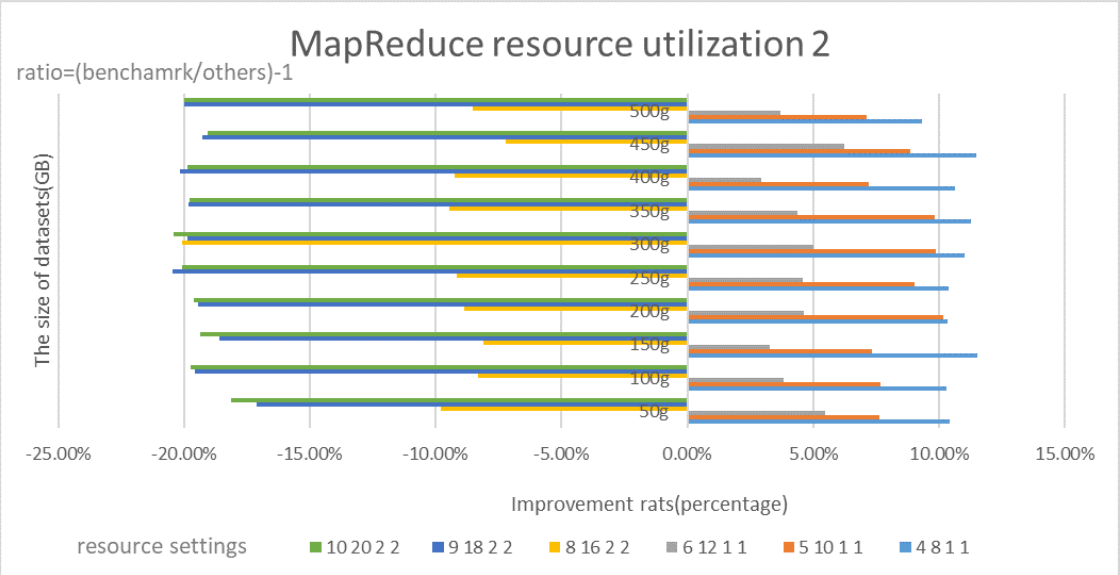

Figure 5.3: The gap graph for MapReduce after applying different resource strategies

Then, Figure 5.4 shows us the situations when applying different resources for Spark. From the figure we can find that the blue line which stands for 50 executors with 8GB memory and 4 vcores is always the best compared to other settings. Since 50 is the smallest executor number in our experiment while 8GB memory is the biggest memory for each executor. Thus, we can find that the small number executors with large resource settings can be an optimal idea when running Spark jobs. Also, we can find that the trends of yellow and navy fluctuate strongly from 150GB to 500GB. The yellow line shows the worst performance at the beginning and keeps increasing to the end while the navy shows its worst performance at 150GB but climbs obviously during the rest of the experiments and finally reach to the second with a small gap to the blue line. Since both yellow and navy belongs to the large executor numbers with few resources and both of them increase sharply when the data sizes grow, we can find that the large executor numbers with small resource strategy is also usable especially for processing the large datasets.

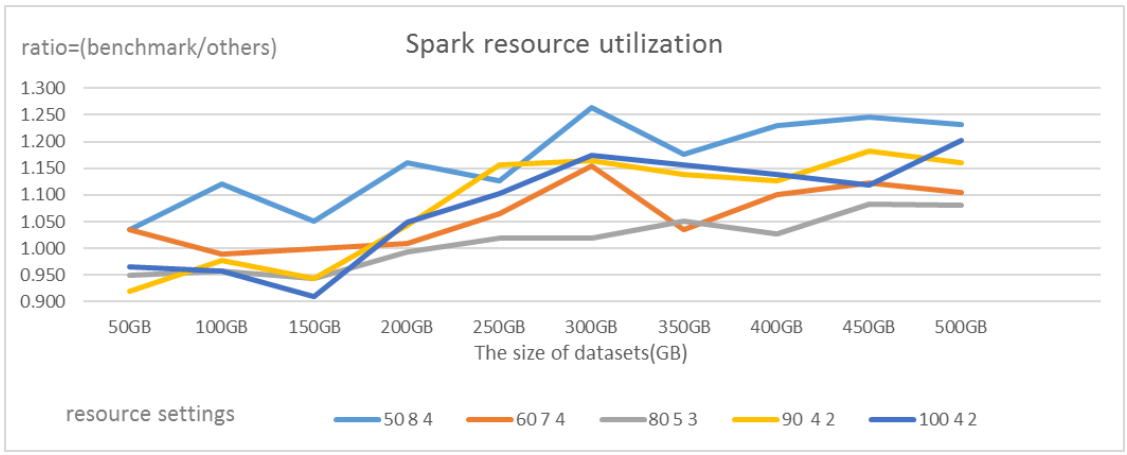

Figure 5.4: The relationship graph of different datasets for Spark

Figure 5.5 presents the optimal improvement rates from resource utilization for different datasets. It divides the datasets as three categories and visualize the improvement under bar chart format. From the figure, we can find that the efficiency of MapReduce jobs improve strongly when processing small datasets while Spark shows higher improvement rates for intermediate and large datasets. Besides, we can find that: for all the datasets, the improvement rates for MapReduce jobs are stable and they are about 10%. For Spark, the rates are small for small datasets and they climb when the datasets grow and finally reach to the top when processing the large datasets. The improvement rates are about twice as much as MapReduce jobs when processing large datasets. Thus, we can find that resource utilization can affect both MapReduce and Spark jobs. However, for MapReduce jobs, tuning resource parameters properly can improve the efficiency slightly for all the datasets while Spark shows high potentiality especially for large datasets.

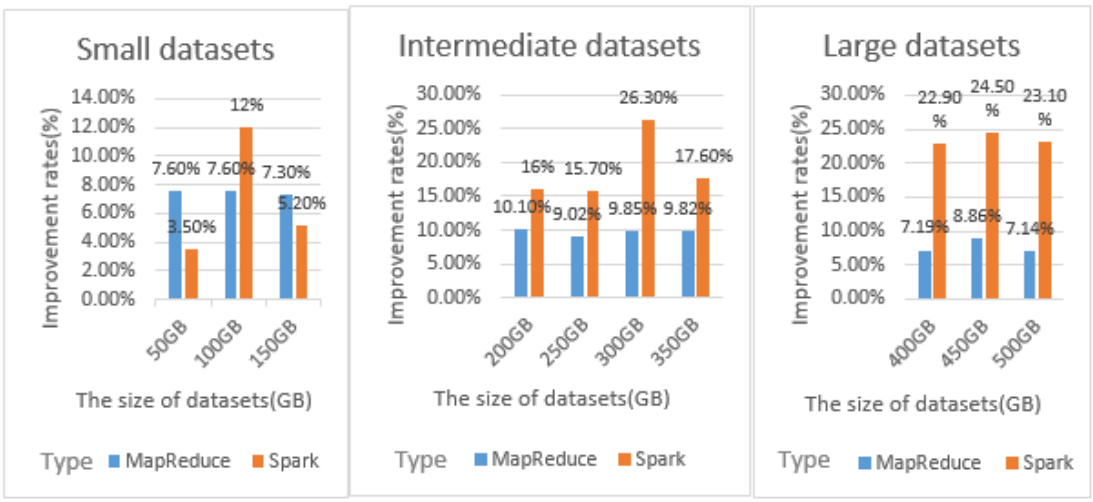

Figure 5.5: The optimal improvement rates for Spark and MapReduce

Finally, Figure 5.6 presents the best execution time after tuning resource parameters for Spark and MapReduce. From the figure, we can find that Spark curve is still sharper than MapReduce curve which means Spark jobs are still more efficient than MapReduce jobs. Also, compared to the Figure 5.1, we can find that the angle between the two lines becomes larger which represents the difference between the two frameworks becomes larger. In other words, Spark is not only more efficient than MapReduce, but also shows higher potentiality for resource parameters tuning.

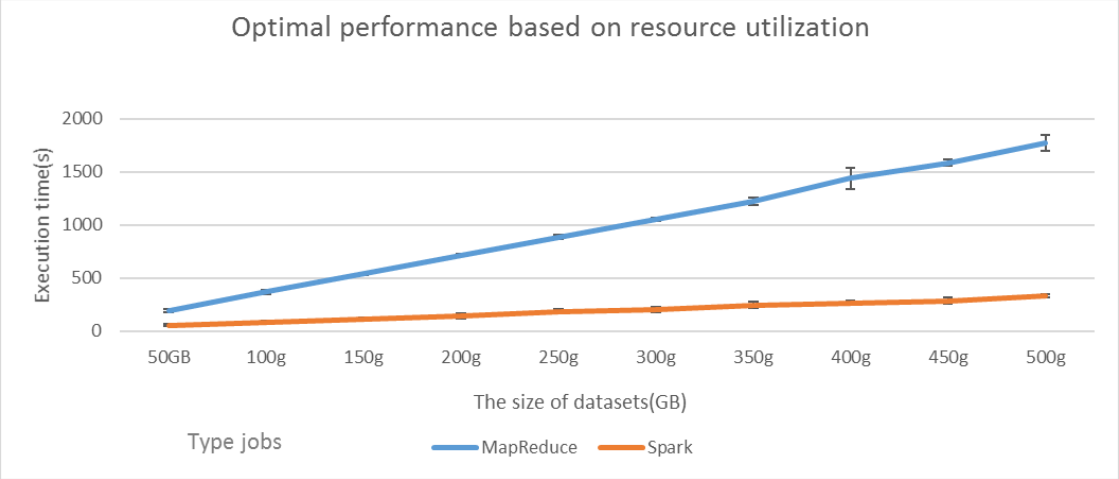

Figure 5.6: The optimal performance for MapReduce and Spark with different resource strategies

5.2.2 Input splits

Next, we take input splits into consideration and try to find out whether the related parameters can affect the results obviously. Figure 5.7 shows us the improvement rates of MapReduce jobs for different input splits compared to the default settings. From the figure, we can find that all the data on the figure is positive which means adding the size of input splits has positive effects on the job efficiency. Also, we can see that the trend of the orange line is always increasing after experienced a drop from the beginning while the trend of the grey line keeps climbing from the bottom and finally reach to the top when the size of datasets reaches to 350GB. Besides, regarding to the improvement rates, they are various when the datasets smaller than 150GB. However, the rates of orange and grey lines gathered at the range between 15% and 20% after the datasets becomes larger. Thus, we can conclude that adding the input splits values properly can affect the efficiency of the MapReduce jobs and may bring a increment about 15% to 20% when processing large datasets.

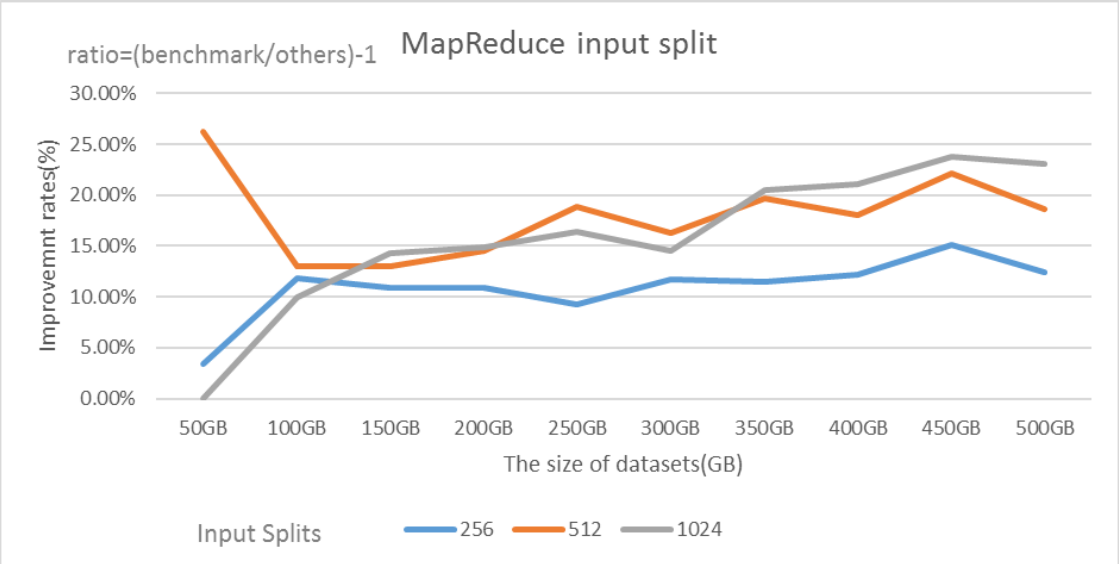

Figure 5.7: The performance of MapReduce jobs with different input splits

Figure 5.8 presents the performance improvement rates compared to the benchmark when applying different input splits. From the figure, we can see that all the data from the figure are negative which means that adding the value of input splits may bring negative effects on Spark jobs. Also, the increase rates are various at the beginning and the smaller input splits shows its advantages before the size of datasets reach to 300GB. After that, all the lines overlap together which means the effects of different settings becomes similar. Thus, we can conclude that adding input splits for Spark is not a wisdom strategy and we need to keep it as the default value (128MB). Adding the value can not only decrease the efficiency and may bring about 25% to 30% side effects to Spark jobs.

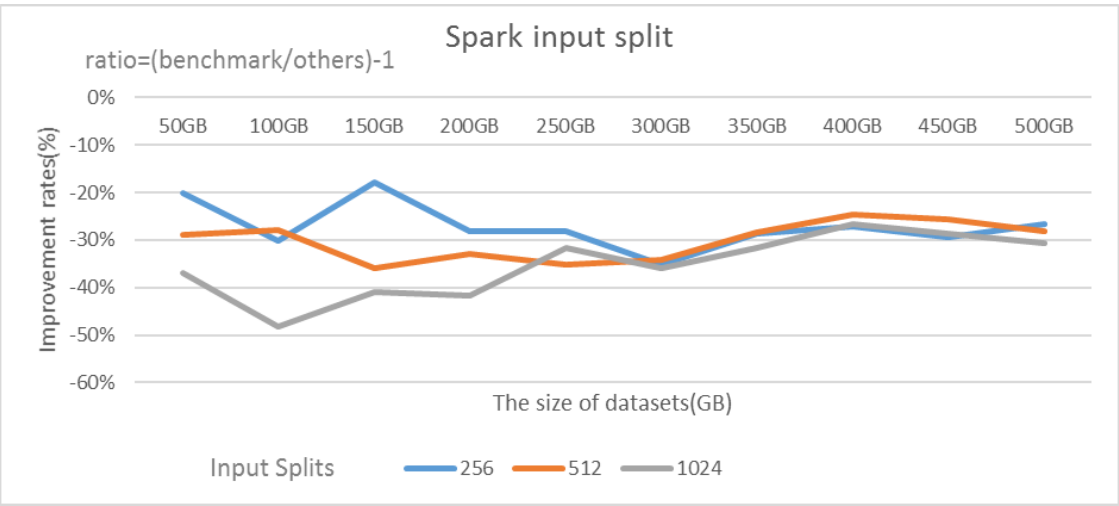

Figure 5.8: The performance of Spark jobs with different input splits

Figure 5.9 and Figure 5.10 illustrate the improvement rates for MapReduce and Spark

in details. From Figure 5.9, we can see that: apart from the first dataset, the trend of the MapReduce jobs increases when the sizes of the datasets grow. For small datasets, the improvement rates are around 15% while the rates increase to 19% for the intermediate datasets. The rates finally reach to 23% for the big datasets. Thus, we can conclude that the input splits can affect the efficiency of the MapReduce job and the improvements becomes more obvious when the datasets increase. Also, from Figure 5.10, we can find that input splits have negative effects on Spark jobs. For small datasets, the decrease rates are around 20% and they reach to 30% when the sizes of datasets become larger. Finally, the rates go back to 25% for large datasets. Thus, we strongly recommend not to change the input splits value when running Spark jobs especially for processing intermediate data.

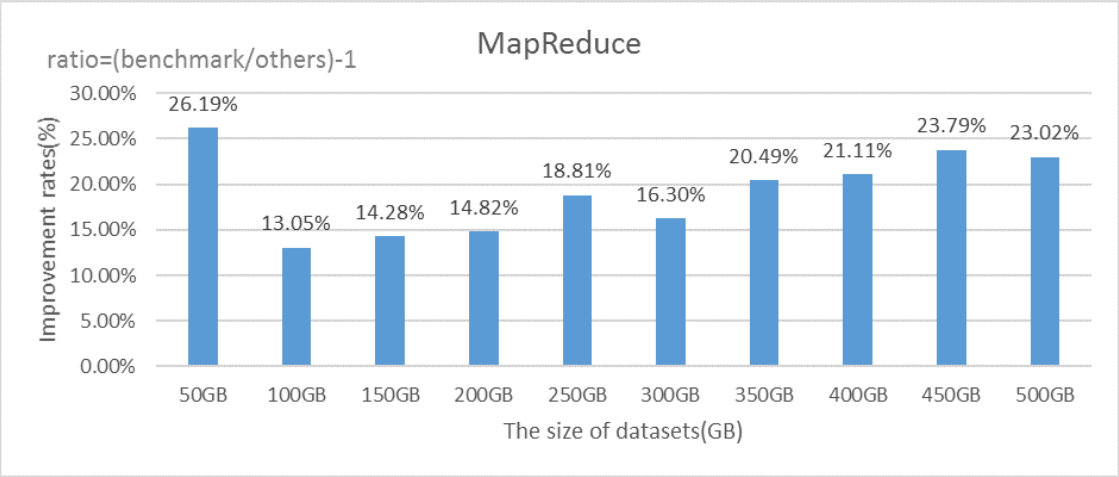

Figure 5.9: The optimal improvement rates for MapReduce with different input splits

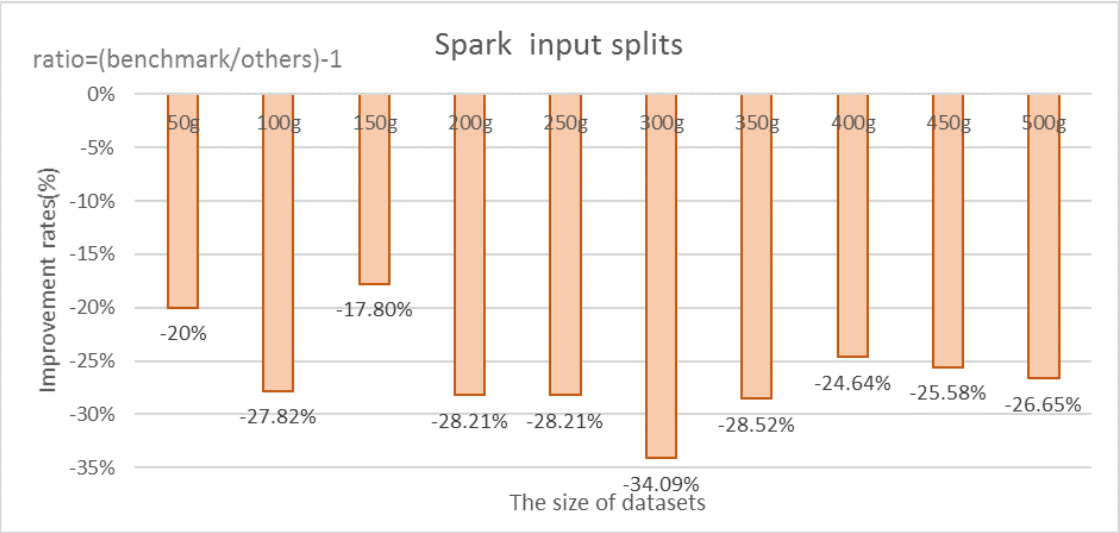

Figure 5.10: The optimal improvement rates for Spark with different input splits

Figure 5.11 illustrates the best performance for MapReduce and Spark after applying

different input splits. From the figure, we can see that Spark jobs shows higher efficiency for all the datasets and its trend is stable with the datasets increases. However, the angle between the two lines becomes smaller which represents that the difference between Spark and MapReduce narrows when applying different input splits. Thus, we can conclude that input splits is a parameter worth to test for MapReduce jobs.

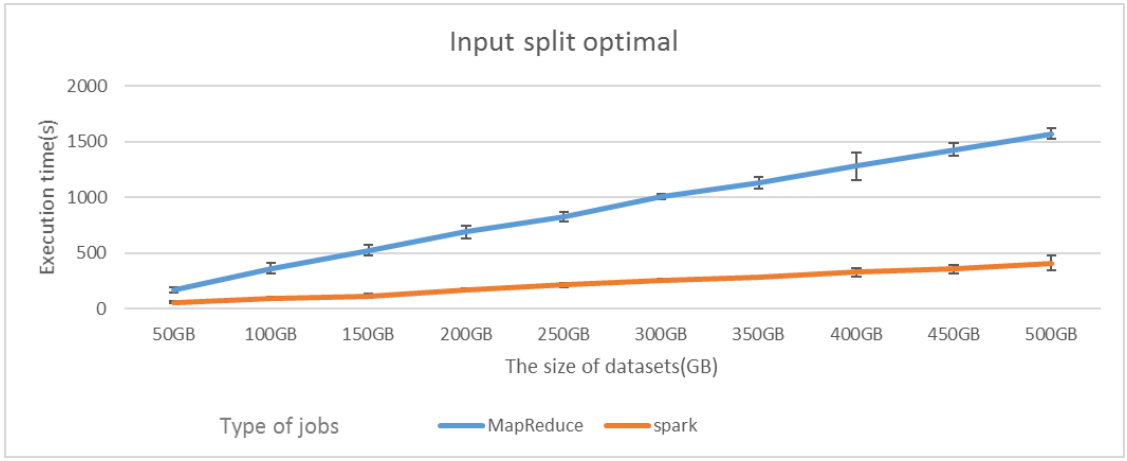

Figure 5.11: The optimal performance for MapReduce and Spark with different input splits

5.2.3 Map-side parameters

Finally, we are going to talk about some other parameters that may affect the MapReduce and Spark jobs efficiency. They are I/O factors and memory for MapReduce jobs and parallelism for Spark. Figure 5.12 shows the MapReduce efficiency with different Map-side I/O settings. From the figure, we can find that grey line which represents the smallest I/O memory and factors presents the best performance and the range of the improvement rates are from 6% to 10%. The blue line and the orange line also present increasing trend and the rates are around 5% and 2%. Thus, we can find out that the Map-side I/O factors can improve the MapReduce efficiency and the smaller I/O settings can be a useful strategy.

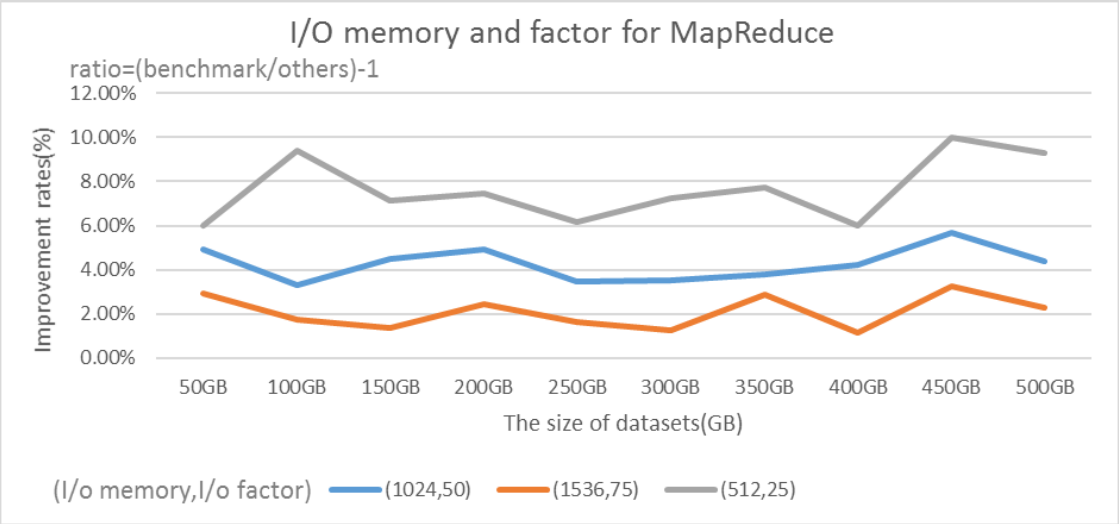

Figure 5.12: The performance of MapReduce jobs with different I/O parameters

Figure 5.13 illustrates the performance changes for Spark jobs with different parallelism degree. Although the situation seems in chaos, the trends are clear and can be divided into three categories: one contains 100,200, another contains 400 and 500 while the others contains 300. For the first category, after experiment a short increase, both lines drop sharply and then keeps climbing to the top with some fluctuation. For the second category, they also increase at the beginning and drops sharply. After that, they climb dramatically to the top and experiment an obvious decrease and finally increase to the end. For the third category, it begins with a moderate increase and then fluctuate to the top. Finally, it experiments a drop until the end. Also, we notice that most of the lines starts with 0 or a negative value which means the parallelism has a small or even negative impacts on the small datasets. However, things change when the datasets become larger since all of the lines increase sharply and keep positive until the end. From the figure, we can find that applying a proper parallelism can improve the efficiency of Spark jobs dramatically. However, the best settings may not be the largest one or the smallest one. Based the trends, we can conclude that we need to set different parallelism degree for different sizes of jobs.

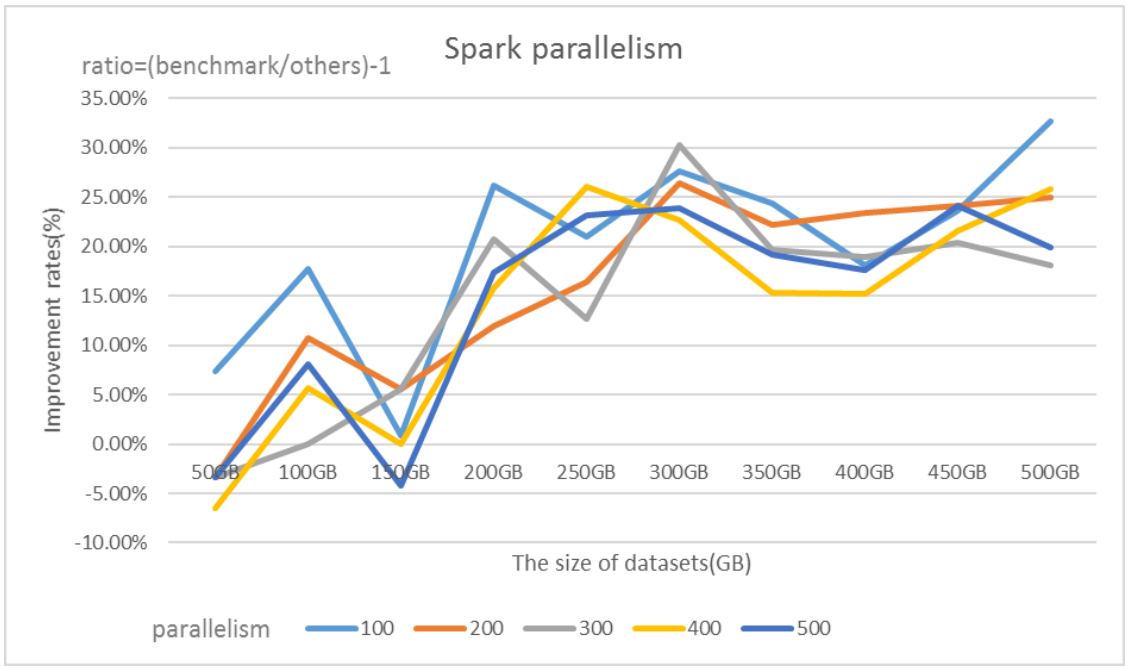

Figure 5.13: The performance of Spark jobs with different parallelism

Figure 5.14 illustrates the best improvement rates for Spark and MapReduce jobs compared to their benchmarks. From the figure, we can find that the improvement rates for MapReduce are stable ranging from 6% to 10% for all the datasets. For Spark, the improvement rates are small for the small datasets while they increase sharply for intermediate datasets around 26%. Then, they keep steady for big datasets around 24%. Based on that, we can conclude that parallelism degree is an important parameter for Spark jobs which can bring an obvious improvement especially for big datasets.

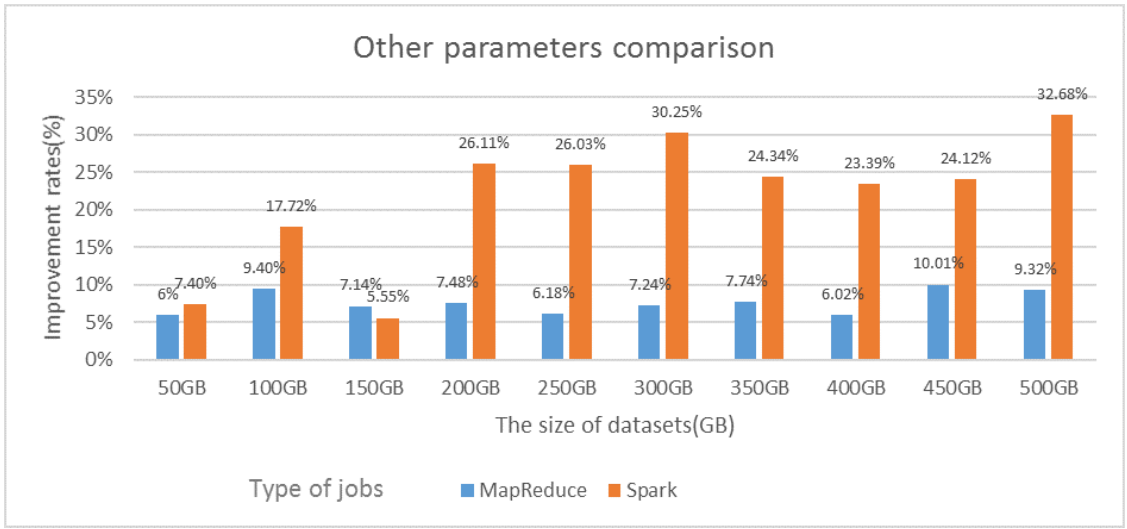

Figure 5.14: The optimal improvement rates for Spark and Mapreduce jobs

Figure 5.15 presents the efficiency difference between MapReduce and Spark jobs after

tuning their own parameters. From the figure, we can find that Spark shows obvious advantages for all the datasets compared to MapReduce and the gap becomes larger when the datasets grows. Also, based on Figure 5.1, we can find that the angle between the two lines becomes smaller which means the difference becomes larger when applying different parameters. Thus, we can conclude that Spark parallelism have stronger effect than Map-side I/O parameters for MapReduce jobs.

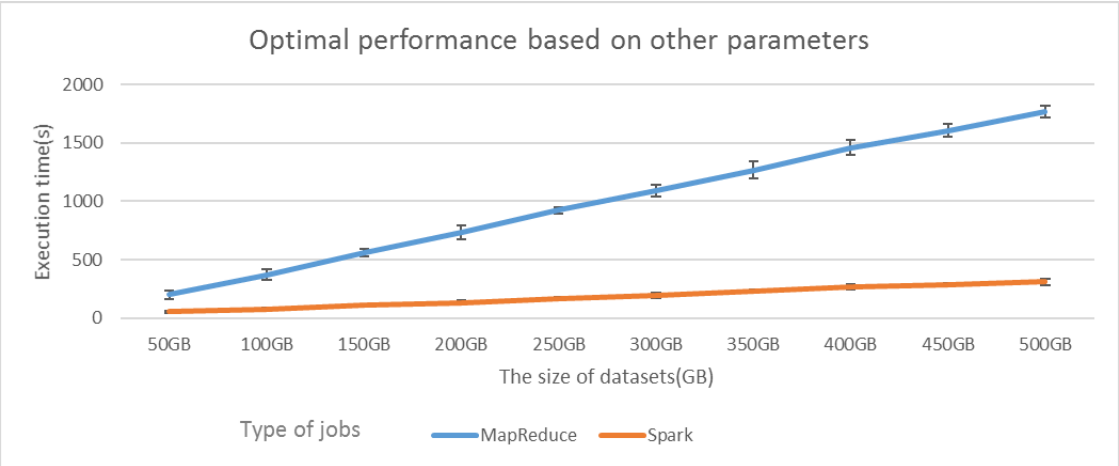

Figure 5.15: The difference for MapReduce and Spark after tuning parameter

5.3 K-means results

The average execution time of K-means can be shown from Table 5.3. The green values represent the worst execution times while the red values stand for the best execution times. Also, the raw data can be found in the appendix. Next, we are going to analysis the results based on the graphs we build.

	50G	75G	100G	125G	150G
MapReduce	591	835	1119	1337	1630
Spark					
Disk_only	194	267	646	944	1554
Memory_only	73	117	140	198	561
(Memory_and_Disk)					
50%	166	262	725	1528	1888
60%	138	240	600	1140	1660
70%	125	221	572	1020	1607
80%	117	202	454	836	1229
90%	111	186	387	693	901
30 executors with 8G memory and 2G memory overhead					

Table 5.3: The result of K-means

5.3.1 MapReduce and DISK_ONLY

Figure 5.16 illustrates the execution time for MapReduce jobs and Spark jobs. From the figure, we can find that there is a huge difference between MapReduce jobs and Spark jobs at the beginning and this difference becomes smaller with the datasets grow. Finally, the gap narrows to the minimum when the datasets increases to 150GB: 76s. Through the result, we can conclude that Spark is better than Mapreduce when processing lightweight iterative tasks with DISK_ONLY strategy. But when the datasets grow larger, both are good options to handle the tasks.

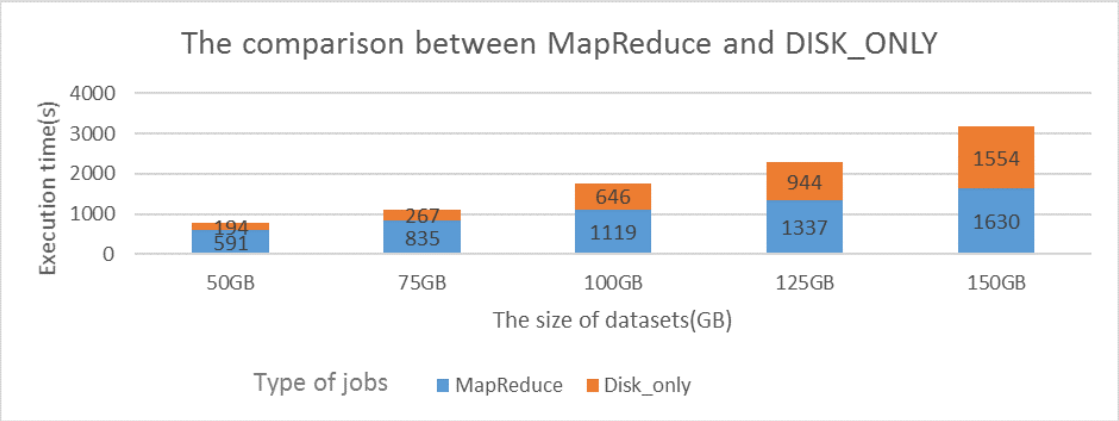

Figure 5.16: The execution time for MapReduce and Spark DISK_ONLY strategy

5.3.2 MapReduce, DISK_ONLY and MEMORY_ONLY

Figure 5.17 presents the situation of MapReduce job, MEMORY_ONLY and DISK_ONLY when processing different sizes of iterative tasks. From the figure, we can see clearly that

MEMORY_ONLY curve appears to a sharp trend which mitigates a little when the size increases from 125GB to 150GB.On the contrary, the curve of DISK_ONLY starts with a sharp trend and experience two mitigation and ends with similar rates as MapReduce curve. For the MapReduce curve, it keeps stable from the beginning to the end. Thus, we can conclude that MEMORY_ONLY strategy shows little advantages for small tasks compared to the other two lines while the advantages becomes more and more obvious with the datasets grow.

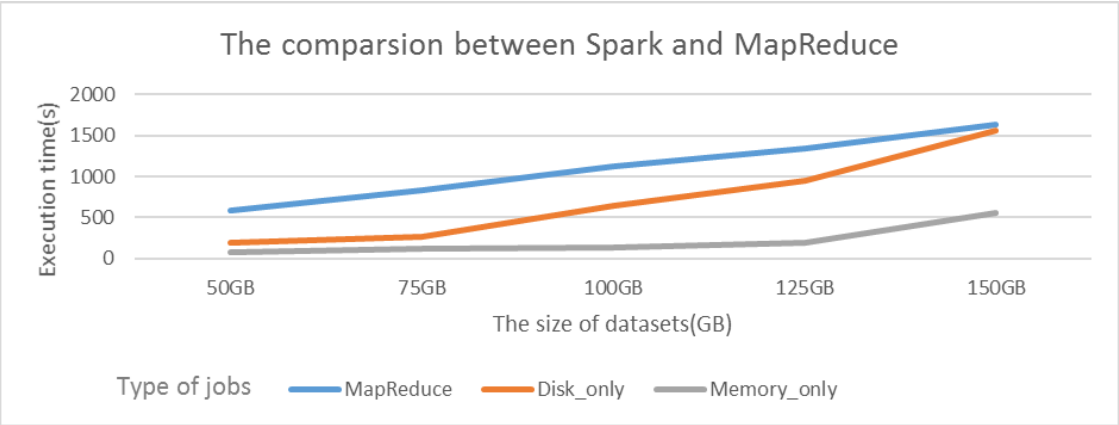

Figure 5.17: The execution time for MapReudce, DISK_ONLY and MEMORY_ONLY

5.3.3 MEMORY_AND_DISK

For MEMORY_AND_DISK strategy, we divide our datasets into two categories: one is small datasets which contains 50GB, 75GB and 100 GB while the intermediate contains 125GB and 150GB. Figure 5.18 shows the execution time for small datasets with different settings. From the figure, we can see that there is a decrease trend when the percentage of memory increases for all the datasets which means the efficiency becomes higher with the datasets increases. For 100GB dataset, the best performance is the strategy with 90% of memory and it can save nearly half of the execution time compared to the strategy with 50% of memory which stands for the worst performance from the figure. Also, there are obvious improvements exit for others when the percentage increases. For 75GB datasets, we can see that the execution time decline sharply compared to 100GB datasets and this improvement trend keeps stable with the percentage increases. For 50GB datasets, the gap is not obvious compared to 75GB datasets and this trend also keep stable until the percentage decrease to 50%. There is a huge decrease happened when the memory rate decreases from 60% to 50%. Thus, we can find that the efficiency of iterative jobs keeps stable when applying DISK_AND_MEMORY strategy for small datasets. However, when the datasets increase to a large number, the efficiency will drop sharply with huge difference between different percentages of DISK_AND_MEMORY value.

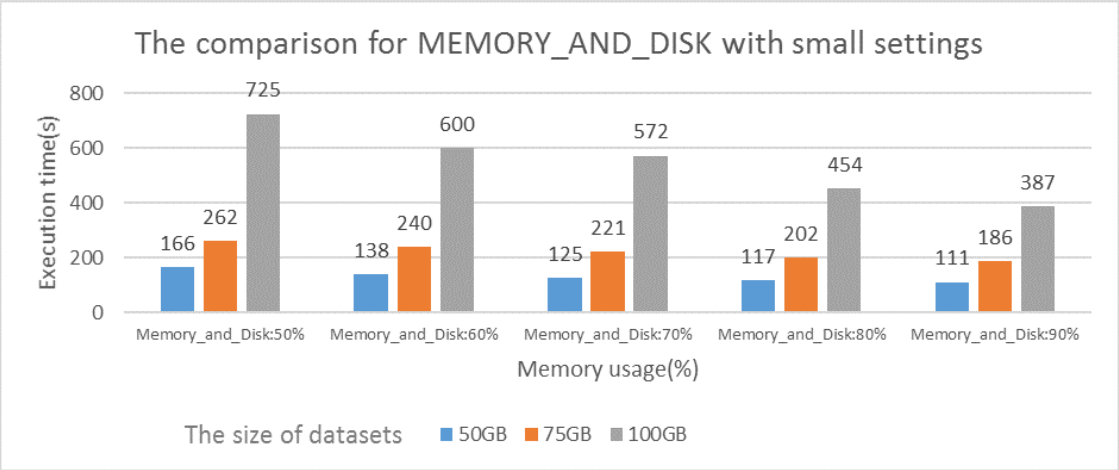

Figure 5.18: The execution time for different settings of MEMORY_AND_DISK

Figure 5.19 belongs to increment graph that mentioned above and illustrates the decrease rates compared to the last execution time when the percentage increases. Thus, the high values present a sharp decrease. Since we select 90% as the benchmark, it is not presented in the picture. From the figure, we can see that all the values are positive which means reduce the memory percentage can have negative effects on the efficiency continuously. Also, we notice that there is an obvious decline during the percentage decrease for the three datasets. For 50GB dataset, it happens when the percentage drop from 60% to 50%. For 75GB dataset, it is from 70% to 60%. For 100GB datasets, it is from 80% to 70%. Since the root causes of the decline are insufficient memory and excessive I/O operation, we can conclude that the memory becomes insufficient during the periods we mentioned above. Also, we find that the situation happens earlier when the datasets grow and the decrease rates of 100GB datasets are much larger than the other two. Thus, we can conclude keeping memory percentage to a higher value is crucial for big datasets to avoid obvious efficiency lose.

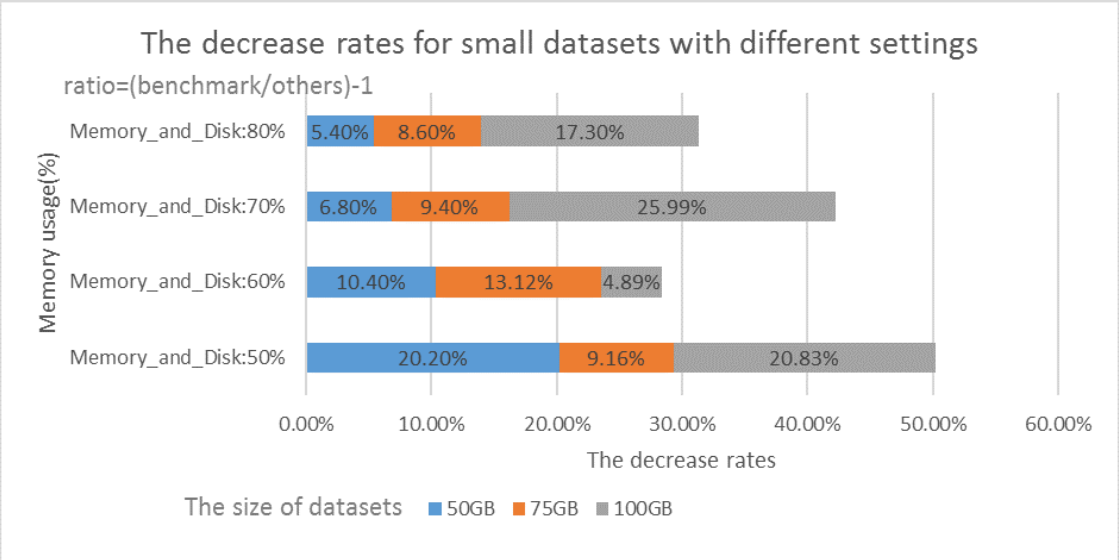

Figure 5.19: The decreasing rates for different settings of MEMORY_AND_DISK

Figure 5.20 presents the results of intermediate datasets. From the figure, we can see that the difference between the best and the worst performance is obvious. For both datasets, the difference reach to 2 times and the gap becomes larger compared to the difference in small datasets.

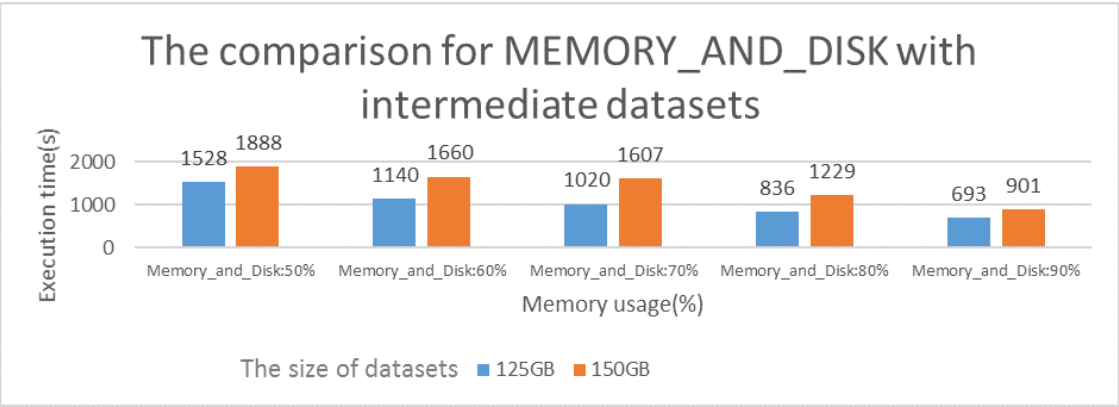

Figure 5.20: The execution time for different settings of MEMORY_AND_DISK

Figure 5.21 belongs to the increment graph and illustrates the decrease rates for intermediate datasets which contains 125GB and 150GB. The benchmark is still set as 90% and thus unable to see it from the picture. From the figure, we can find that the most obvious decline happens from 60% to 50% for 125GB datasets and from 90% to 80% for 150GB datasets. Although this situation happens late for 125GB, the decreasing rates from 90% to 80% and 80% to 70% are still obvious which are 20.63% and 22.00% respectively. Besides, 150GB datasets experience two sharp decrease when the memory percentage decrease from 90% to 70% and the rates are 36.43% and 30.75%. Thus, based

on the findings from the small datasets and intermediate datasets, we can conclude that the memory percentage should be set as a high value for big datasets while it can be set as a low value for some small datasets.

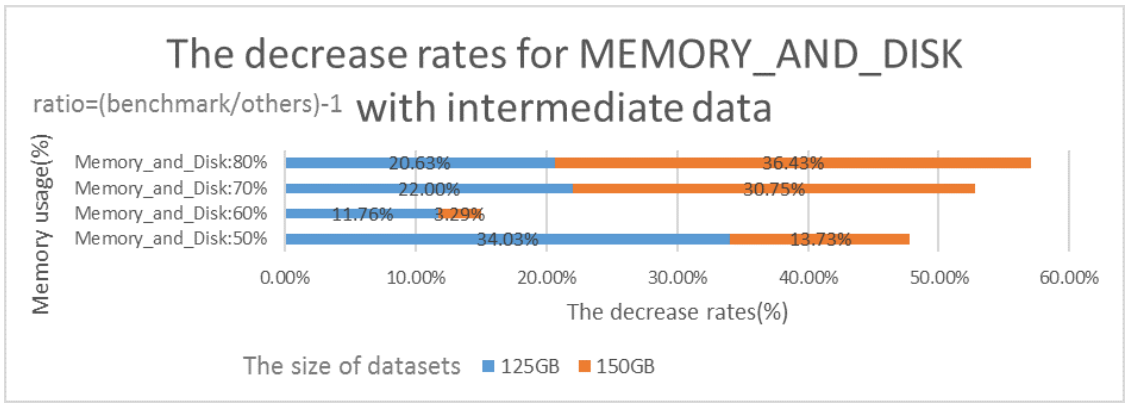

Figure 5.21: The decreasing rates for different settings of MEMORY AND DISK

5.3.4 DISK_ONLY, MEMORY_ONLY and MEMORY_AND_DISK

Figure 5.22 presents the situation for DISK_ONLY, MEMORY_ONLY and best and worst performance of MEMORY_AND_DISK. It is used to help us understand the difference between the three persist strategies provided by Spark. From the figure, we can find that all the curves share the similar execution time around 200s at the beginning. Then, the trend of MEMORY_ONLY increases smoothly all the time until meets a transition when the datasets increase to 150GB. For MEMORY_AND_DISK, the best and worst curves increase with same rates at first and appear different situation when the datasets reach to 75GB. The worst one experience threes obvious mitigation while the best one experience two slight mitigation when processing the following datasets. For DISK_ONLY curve, it overlaps the worst one at first and then experience a slight mitigation followed by an obvious one until the end. Thus, we can conclude that MEMORY_ONLY is the best option for all the iterative tasks. Also, MEMORY_AND_DISK with high memory percentage can bring high efficiency than DISK_ONLY and thus can be the second choice. Finally, DISK_ONLY can be the third choice and much faster than the MEMORY_AND_DISK with low memory percentage.

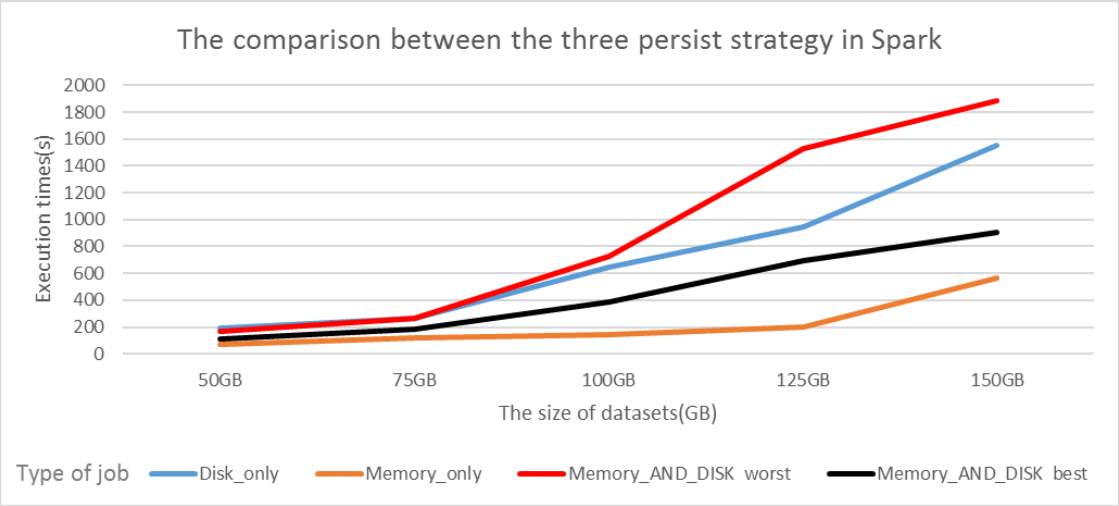

Figure 5.22: The comparison between DISK_ONLY, MEMORY_ONLY and MEMORY_AND_DISK

5.4 TeraSort results

The average execution time of TeraSort for Spark and MapReduce can be seen from the Table 5.4 and Table 5.5. The green values stand for the worst execution time while the red values represent the best based on different settings. Also, the raw data can be seen in the appendix. Next, we are going to analysis our results based on the graphs we build.

Hadoop	50G	100G	150G	200G	250G	300G	350G	400G	450G	500G
Resource utilization										
14G 1vcore:reducer 30 :reduce number (default)	190	346	619	987	1076	1779	1919	2011	2482	2779
10G 1vcores: reduce 40:reduce number	230	422	847	1268	1471	1786	2186	2678	2886	2914
12G 1vcores: reduce 35:reduce number	221	405	745	1229	1315	1741	2075	2265	2503	2795
16G 2vcores: reduce 25:reduce number	214	374	582	825	1079	1350	1611	2024	2461	2723
18G 2cvore: reduce 20:reduce number	205	327	556	813	1023	1223	1457	1734	2058	2412
Input Splits:										
128M(default)	190	346	619	987	1076	1779	1919	2011	2482	2779
256M	143	316	486	705	930	1217	1680	1946	2393	2705
512M	187	269	466	660	903	1255	1514	2170	2561	2837
1024M	182	319	540	705	1042	1304	1560	1969	2780	2993
Reduce-Side parameters										
default(100,30)	190	346	619	987	1076	1779	1919	2011	2482	2779
mapreduce.reduce.shuffle.parallelcopies=200 mapreduce.task.io.sort.factor =60	184	415	612	894	1169	1719	1911	2394	3032	3429
(150,45)	176	372	562	813	1175	1501	1998	2267	2880	3594
(75,20)	185	350	608	963	1133	1482	1996	2273	2785	3109
(50,15)	189	358	644	871	1242	1619	1901	2495	2718	2884

Table 5.4: The result of MapReduce TeraSort

Spark	50G	100G	150G	200G	250G	300G	350g	400G	450G	500G
Resource utilization (70 executors 6G memory 3 vcores)										
Reducer number:20	287	584	961	1261	1712	2211	2584	3113	3258	3656
25	253	505	899	1183	1588	2043	2467	2651	2869	3096
30(default)	239	496	857	1093	1444	1826	2067	2445	2794	3042
35	231	435	677	1051	1239	1764	1947	2019	2529	2839
40	232	425	619	984	1222	1531	1698	1967	2475	2750
Input splits (reducer number:30)										
128M(default)	239	496	857	1093	1444	1826	2067	2445	2794	3042
256M	266	457	682	919	1371	1612	2052	2147	2507	2928
512M	252	491	792	1101	1451	1845	2072	2188	2806	3228
1024M	297	532	848	1215	1465	1895	2309	2695	2989	3151
Shuffle parameters										
spark.shuffle.file.buffer=16k spark.reducer.maxSizeInFlight=24M	233	501	849	1135	1509	1610	2131	2486	3007	3080
(32k,48M)	212	492	750	1084	1496	1565	2006	2446	2621	2796
(64k,96M)	238	471	744	920	1289	1439	1909	2138	2434	2731
(128k,192M)	193	444	673	947	1191	1400	1675	1990	2410	2634
Replication of Spark jobs =1										

Table 5.5: The results of Spark Terasort

5.4.1 Number of reduce tasks

Firstly, we are talking about the number of reduce tasks for MapReduce and Spark. For both frameworks, the default number of reduce tasks will be one which is unable to fully utilize our cluster resource and thus reduce the job efficiency. For MapReduce jobs, we keep the default resource settings of the reduce tasks as 14GB memory and 1 vcore and set the number of reduce tasks as 30. For Spark, we launch 70 executors and give each of them 6GB memory and 3 vcores and set the number of reduce tasks as 30. Then, we set both as the benchmark to help us present our finding easily. The figures below present our results.

Figure 5.23 illustrates the comparison between the two benchmarks. From the figure, we can find that MapReduce job is always superior to Spark jobs. Besides, The Spark curve shows a stable trend during the whole process and the execution time grows steadily with the datasets becomes larger. However, MapReduce curve experience two mitigations when the datasets reach to 200GB and 300GB while it stills shows higher efficiency at the end of our experiment. Based on the figure, we can conclude that MapReduce is better than Spark when processing the shuffle jobs before tuning any parameters.

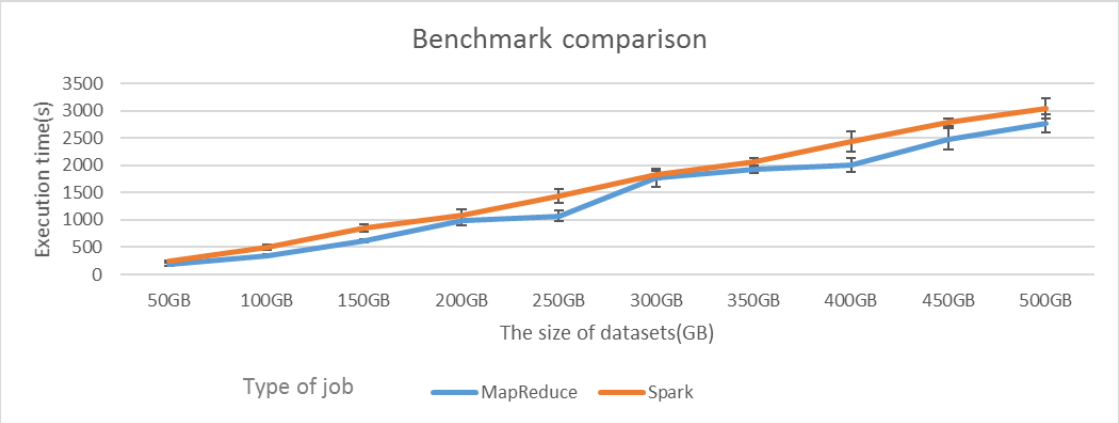

Figure 5.23: The comparison between the benchmarks for Spark and MapReduce

Figures 5.24 and 5.25 present the relationship and gaps after changing the number of reduce tasks for MapReduce jobs. From Figure 5.24, we can find that there are two curves that are always above 1.0 which means that these two are always superior to the benchmark. On the contrary, the orange and blue curves experienced some fluctuations and finally failed to beyond 1.0. Besides, the yellow and grey curves can represent the strategy with few reduce tasks and more resource while the other two represent large reduce tasks with few resource. Thus, we can conclude that the less reduce tasks with more resource can be a wisdom strategy and can bring positive effects to the MapReduce jobs compared to the benchmark. From figure 5.25, we can find the specific improvement rates for MapReduce tasks after configuring different reduce tasks. At the beginning, all the values are negative until the datasets reach to 100GB. Then, the grey and yellow pillar keeps positive while the blue and orange pillar keeps negative. Based on the fact, we can find that the difference between the two is obvious at first and reach to the top when the datasets reach to 300GB and then the difference narrows until the end. Finally, we can conclude that the number of reduce tasks can strongly affect the efficiency of MapReduce jobs especially for small and intermediate datasets.

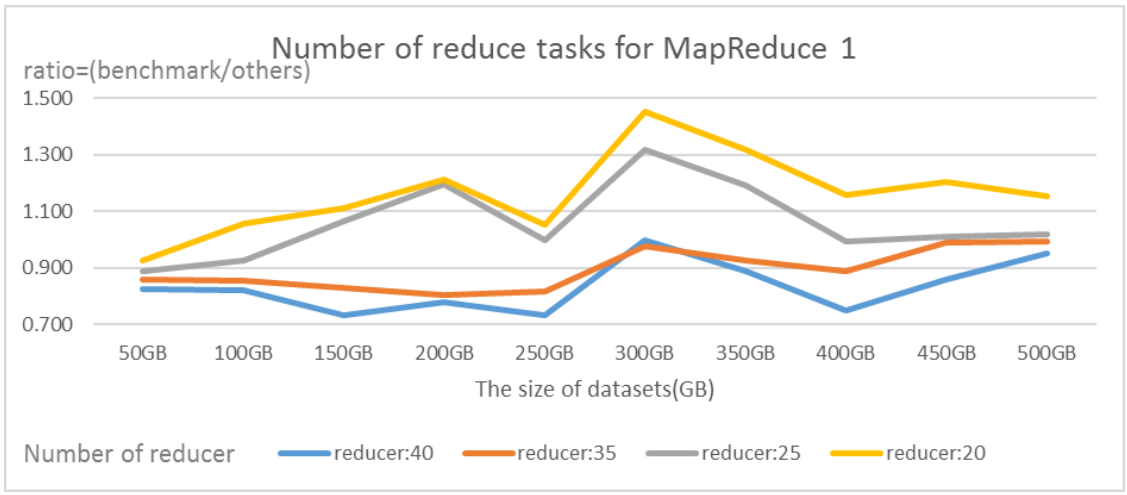

Figure 5.24: The relationship graph for MapReudce after changing the number of reduce tasks

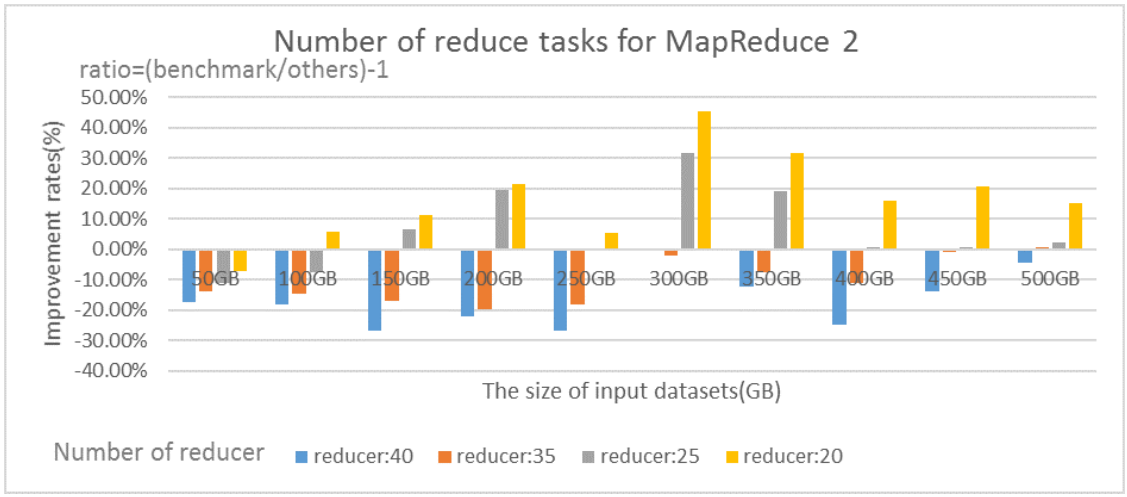

Figure 5.25: The gap graph for MapReudce after changing the number of reduce tasks

Figures 5.26 and 5.27 illustrate the relationship and gap graphs for Spark after applying different numbers of reduce tasks. From the first figure, we can find that the blue and orange curves are above 1.0 while the orange and grey curves are below. Also, since Spark provides the fixed resource before executing the experiments, the orange and blue line can be recognized as the higher efficiency compared to the benchmark while the grey and yellow represent the lower efficiency. Thus, we can conclude that the few reduce tasks under fixed resource is a good option for Spark jobs which can bring obvious improvement. From the second figure, we can find that blue and orange pillars keep positive while the grey and yellow pillars keep negative. Also, the biggest difference between the two happens when the datasets reach to 150GB. Then, the difference keeps steady and becomes a little bit smaller at the end of our experiment. In fact, when the datasets reach to 400GB,

the blue and orange curves show an obvious decrease and the other two present a slightly increase. Thus, we can conclude that the number of reduce tasks can strongly affect the efficiency of Spark jobs and the effects becomes obvious from intermediate datasets and becomes less obvious when the datasets becomes larger.

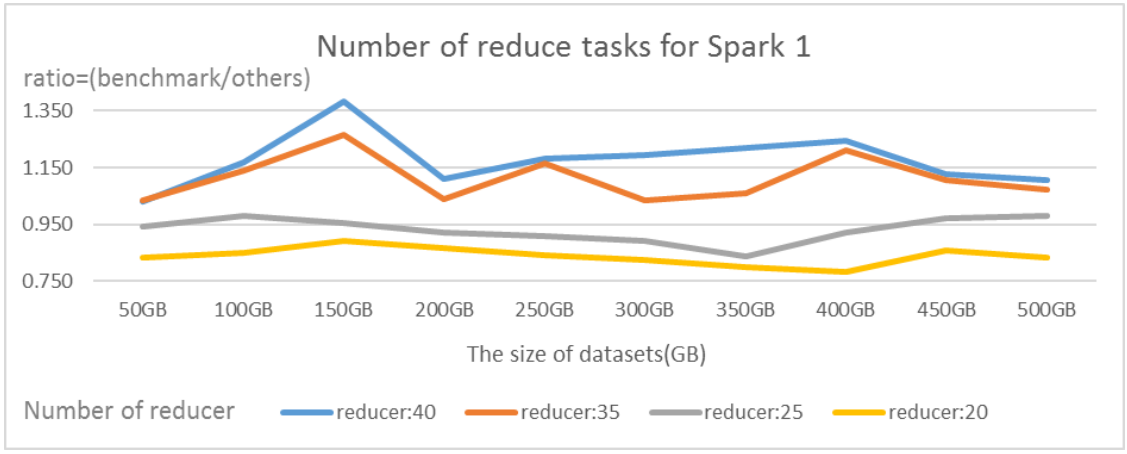

Figure 5.26: The relationship graph for Spark after changing the number of reduce tasks

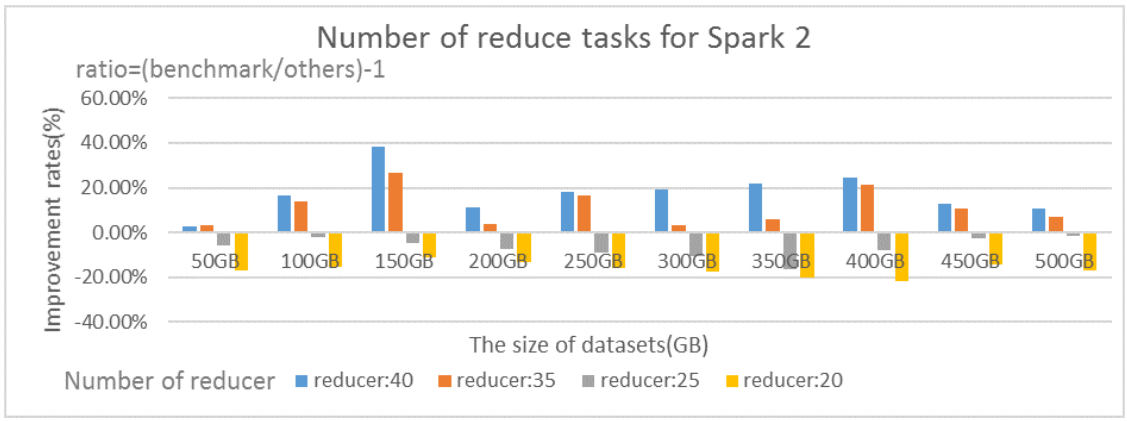

Figure 5.27: The gap graph for Spark after changing the number of reduce tasks

Figure 5.28 shows the best performance for MapReduce and Spark after configuring the number of reduce tasks. From the figure we can find that MapRede jobs are still superior to Spark jobs when processing the shuffle tasks. Compared to the benchmark comparison graph, we can find that the difference for small datasets is nearly disappear while the difference becomes more and more obvious when the datasets reach to 200GB. Thus, we can conclude that MapReduce jobs is better than Spark jobs when applying different number of reduce tasks and its parameters brings higher efficiency than Spark parameters do.

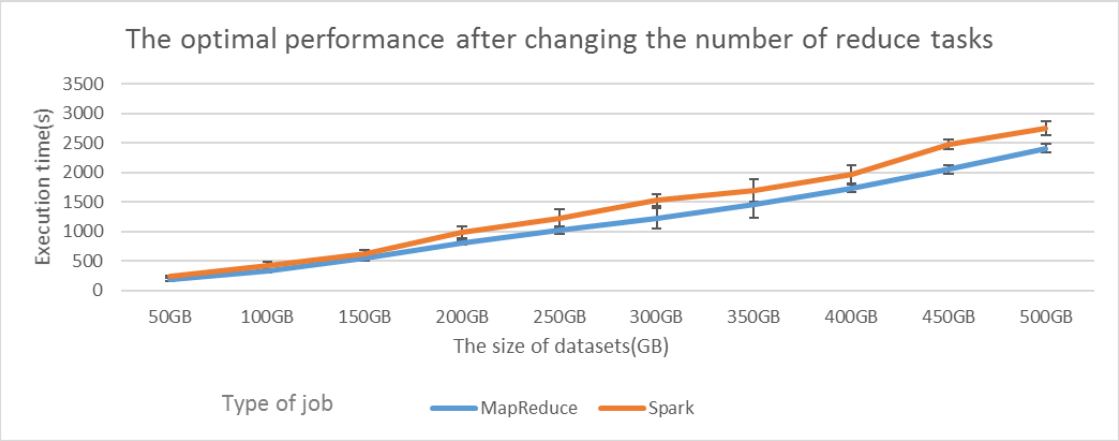

Figure 5.28: The optimal performance for MapReduce and Spark after tuning number of reduce tasks

5.4.2 Input splits

Next, we take input splits into consideration. Since the default input splits for Spark and MapReduce is 128MB, we set these experiment results as the benchmark and compare them with other settings. Figure 5.29 presents the gap graph after tuning the parameters related to the input splits. From the figure, we can find that the curves are always positive which means the input splits can bring positive effects to the MapReduce jobs. Also, we notice that the orange and blue curves show the similar trend from 100GB and they can bring similar improvement rates to the jobs. On the contrary, the trend of grey curve is not as sharp as the other two and ends at the negative rates which will bring negative effects compared to the benchmark. Besides, since the orange and blue represent the small and intermediate value for the input Spark jobs while the grey one represents the large value, we can conclude that increase the value of input splits can be a good suggestion for shuffle jobs and the value should be not too large.

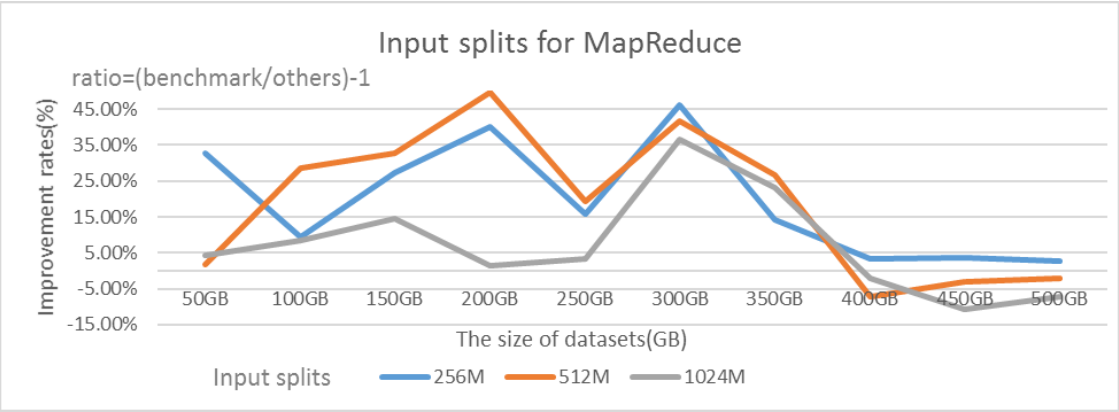

Figure 5.29: The gap graph after tuning input splits for MapReduce jobs

Figure 5.30 illustrates the gap graphs for Spark jobs with different input splits. From the figure, we can find that the blue and orange curves always stay positive while the grey curve keeps negative compared to the benchmark. For the blue curve, the trend is sharp at the beginning and fluctuate to around 5% in the end. The orange curve starts with a negative value experience two fluctuations and a stable period and finally drop to -5%. The grey line starts with an extreme negative value and climb slightly to a similar value as the orange curve. Also, since the three curves represent the small, intermediate and large value of input splits for Spark, we can conclude that changing the input splits may not a compulsory to improve the Spark efficiency and increase this value slightly can bring positive effects.

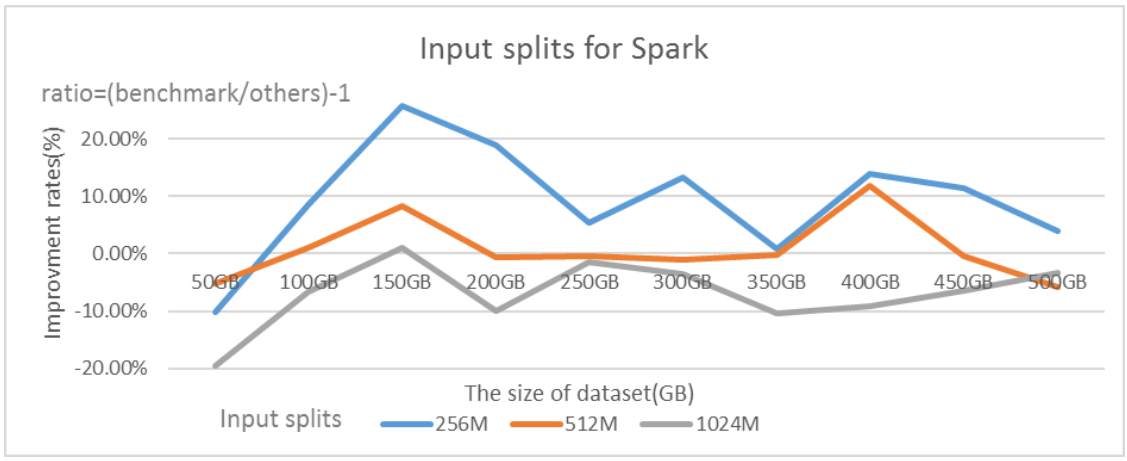

Figure 5.30: The gap graph after tuning input splits for Spark jobs

Figure 5.31 express the optimal improvement rates after configuring the best value of input splits. From the figure, we can find that the improvement rates for MapReduce are high until the datasets increase to 400GB and the Spark jobs shows small improvement at first while increase sharply when processing the big datasets. For MapReduce jobs, we can see that the improvement rates are high and the best performance happens at 200GB which can bring nearly 50% improvement. However, this situation ends when the datasets reach to 400GB and the parameters can bring about 3% improvement for the following experiments. For Spark jobs, the improvement rates are stable compared to MapReduce jobs at first and the difference is obvious from the graph. However, when the datasets increase to 400GB, the improvement rates for Spark beyond MapReduce jobs and about several times as much as the MapReduce jobs. Thus, we can conclude that input splits should be considered as an important factor to improve the efficiency of MapReduce shuffle jobs when processing the small and intermediate datasets while it is more important for Spark when processing large datasets.

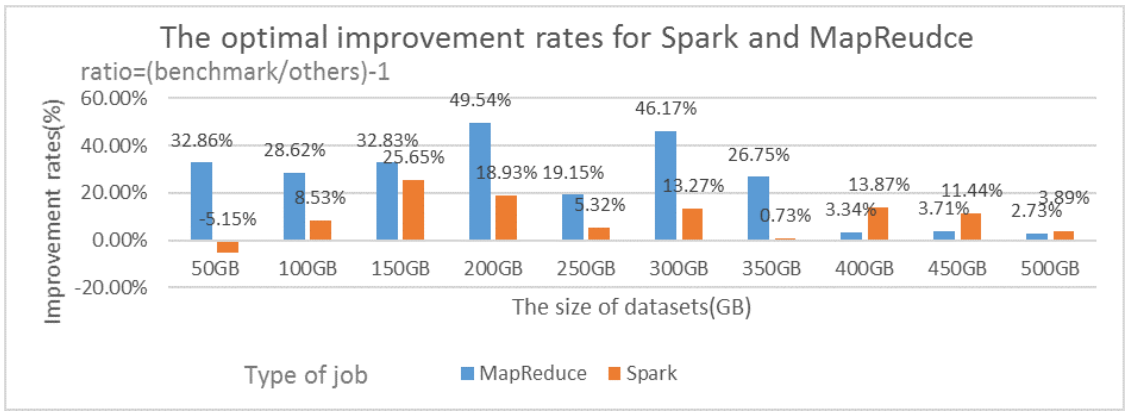

Figure 5.31: The optimal improvement rates for MapReduce and Spark

The Figure 5.32 presents the difference after configuring the best input splits for MapRedue and Spark. From the figure, we can find that MapReduce jobs are still superior to the Spark jobs. However, compared to the benchmark graph, the gap narrows at the beginning and the end while the difference becomes larger from 200GB to 350GB. Thus, we can make a conclusion that MapReduce jobs are more efficient than Spark jobs for shuffle tasks after configuring the input splits and the parameters bring different effects for different sizes of data.

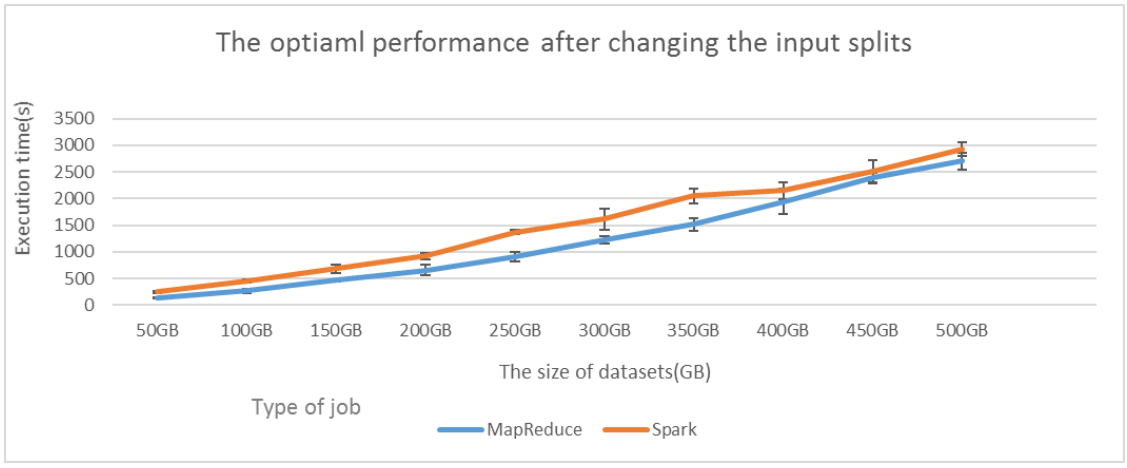

Figure 5.32: The Comparison after configuring the best input splits for Spark and MapReduce

5.4.3 Shuffle-related parameters

Then, we take some shuffle-related parameters into consideration. For MapReduce jobs, the default *mapreduce.reduce.shuffle.parallelcopies* is 100 and *mapreduce.task.io.sort.factor* is 60. For Spark, *spark.shuffle.file.buffer* and *spark.reduce.maxSizeInFlight* are not included in the default settings. Thus, we set the MapReduce jobs with the default settings and the Spark jobs without any settings as the benchmark for further comparison. Figure

5.33 illustrates the gap graph after configuring different reduce-side parameters. From the figure, we can find that the trends for all the settings are similar and half of the experiment results keeps positive while the others are negative. Also, we find that there are two periods that the positive effects are obvious which are 100GB to 250GB and 250GB to 350GB. Besides, the period between 400GB to 500GB are always negative. As a result, we can conclude that the reduce-side parameters are suitable for small and intermediate datasets when processing the shuffle jobs under the MapReduce framework.

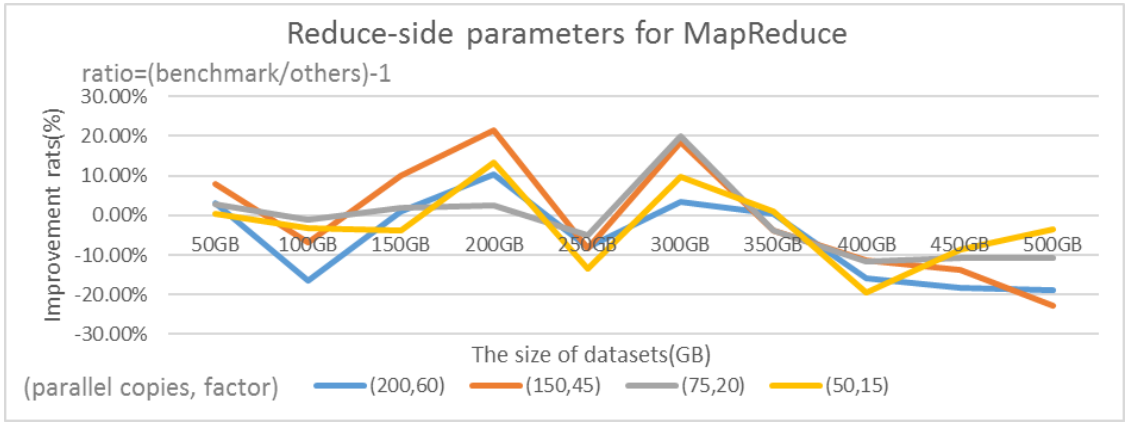

Figure 5.33: The gap graph after changing the reduce-side parameters

Figure 5.34 present the gap graph after adding the shuffle-related parameters for Spark jobs. From the figure, we can find that yellow, orange and grey curves keep positive while the blue curve after a slight decrease and a sharp increase keeps negative at the end. Also, the trends for yellow is the most sharp followed by the grey and orange. Since the yellow and grey curve can be recognized as big value group and the blue and orange can be reckoned as the small value group, we can conclude that adding the shuffle-related parameters can improve the job efficiency for Spark under most circumstances and the large value should be a better option.

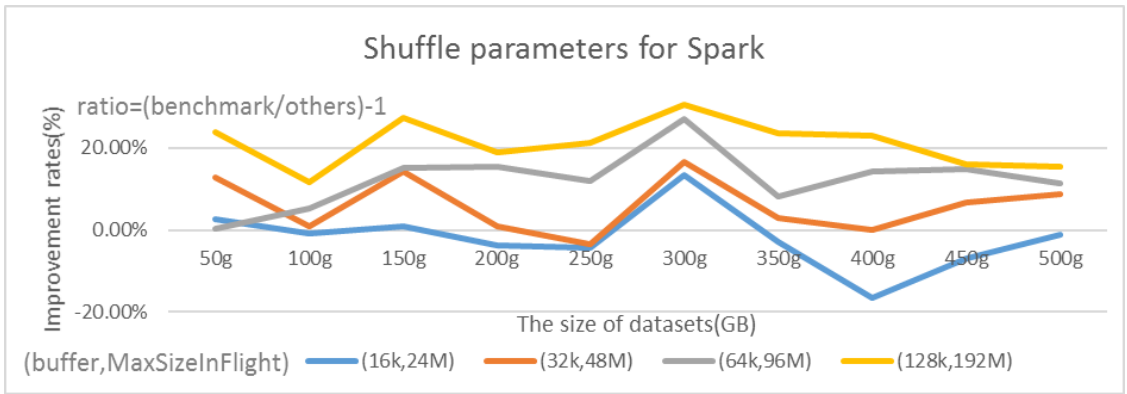

Figure 5.34: The gap graph after changing the shuffle-related parameters

Figure 5.35 presents the improvement rates after configuring the best shuffle-related parameters for Spark and MapReduce. From the figure, we can find that Spark parameters show higher efficiency for almost all the datasets and the improvement rates are obvious. On the contrary, MapReduce parameters fluctuates strongly and the best period should be intermediate datasets. Thus, we can conclude that shuffle parameters related to Spark are strongly suggested to improve efficiency while the parameters related to MapReduce shuffle process need to be considered seriously.

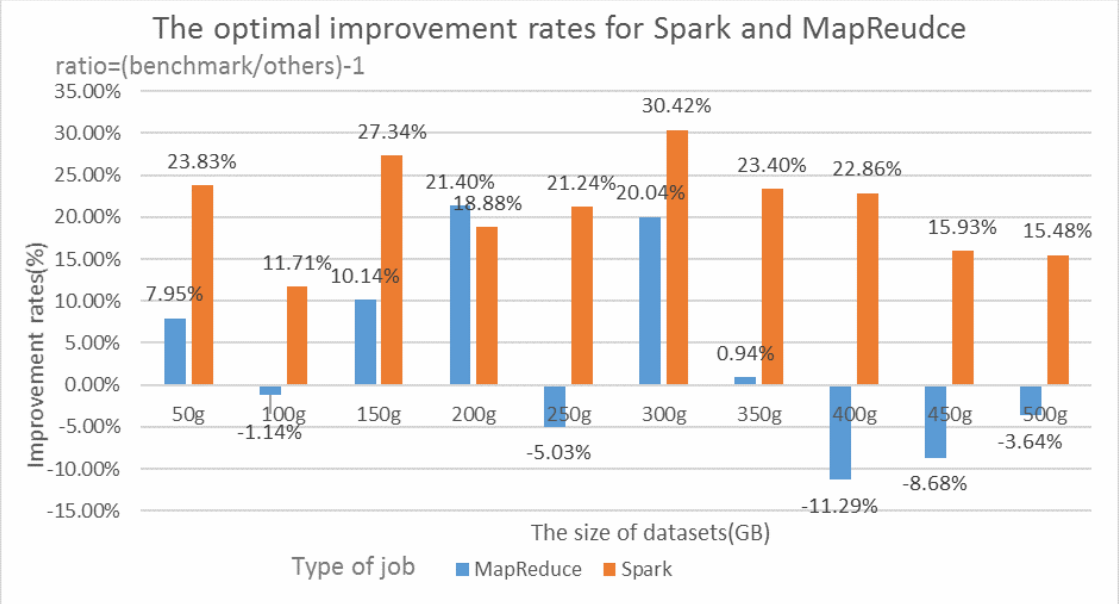

Figure 5.35: The optimal improvement rates after changing the shuffle-related parameters

Figure 5.36 illustrates the performance difference after changing the parameters related to the shuffle process. From the figure, we can find that two curves intersect together which represents the difference between MapReduc and Spark is not obvious. Also, we find that MapReduce shows higher efficiency until 250GB and then Spark overtakes it and finally win the match. Compared to the benchmark graph, we can notice that Spark is superior to MapReduce after tuning the parameters related to the shuffle process. Finally, we can conclude that parameters related to Spark can bring obvious improvement to the jobs and thus strongly suggested to be configured.

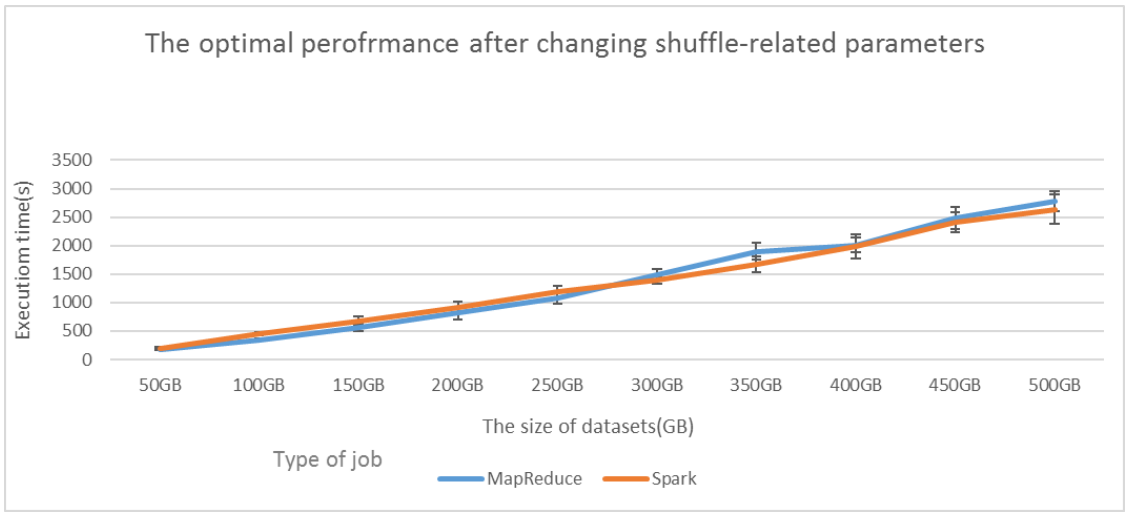

Figure 5.36: The difference after configuring the best shuffle-related parameters

Chapter 6

Insights and suggestions

In this section, we conclude the insights from our experiment results and discuss the lessons we learned from our research. Overall, the result shows that Spark is more efficient when processing aggregation and iterative jobs while MapReduce presents its advantages when facing the shuffle jobs. Although it is a well-known fact that Spark is faster than MapReduce under most circumstances, our thesis presents a novel analysis of the performance difference between different kinds of jobs and take various parameters into consideration. Particularly, we compare the original difference between Spark and MapReduce jobs with default settings as well as the difference after tuning the parameters. By plotting different graphs, we visualize the difference and present it vividly to the users. Also, through the experiments, we gained lots of experience about running different kinds of jobs which can help us provide the suggestions for the users when implementing these kinds of jobs. In the next section, the details of the findings are going to be discussed.

6.1 Wordcount

For WordCount or similar workloads, Spark presents a comprehensive advantage from all the aspects. For the jobs with default settings, Spark jobs are slightly faster than MapReduce jobs when processing the small datasets while the difference becomes huge when the datasets grow to intermediate or large group. Also, when considering the resource parameters for both jobs, MapReduce are more sensitive when the datasets are small and the improvement rates keep around 10% for all the datasets. Spark shows a little improvement for the small datasets and keep increasing until the datasets reach to large group. The improvement rates for small, intermediate and large groups are about 5%, 17% and 23% respectively. Thus, the difference after tuning resource parameters become larger for MapReduce jobs.

Considering the input splits, Spark and MapReduce show the opposite results when processing the different datasets. For MapReduce jobs, it presents the positive effects and

the larger value can bring the significant improvement when the datasets grow. However, Spark shows a decrease trend when enlarging the input splits. These parameters weaken the efficiency in varying degrees when process the small datasets while the side effects become same when the datasets grow to large degree.

The last aspect is map-side parameters for MapReduce and parallelism for Spark. For MapReduce jobs, keep the settings small can bring positive effects and the optimal improvement rates is about 7%. For Spark jobs, the parallelism brings negative effects when the datasets are small while the trend reverses and reach to the top when the datasets grow from intermediate to large group. The optimal improvement rates are about 25% for intermediate and large group. Thus, the difference after tuning these parameters become larger for MapReduce jobs.

In conclusion, all the aspects can bring positive effects for MapReduce jobs through our experiments. The average optimal improvement rates are 10.65% for resource utilization,19.18% for input splits and 7.65% for map-side parameters. Based on that, we can provide the following suggestions that: for the users who are running the MapReduce WordCount or related workloads:

(1) Reducing the memory and vcores to a small value for both map and reduce tasks can dramatically improve the MapReduce efficiency.

(2) The I/O related parameters for MapReduce can slightly improve the efficiency and smaller value shows better performance.

(3) Enlarging the input splits is a good strategy when running the WordCount and the larger value can strongly affect the efficiency especially for big datasets.

For Spark jobs, through our experiments we can find that: resource utilization, parallelism can improve the efficiency while the input splits can reduce the efficiency. Also, the average optimal rates for different aspects are: 16.68% for resource utilization, -25.45% for input splits and 21.75% for parallelism. Thus, we can give the suggestion that for the users who run Spark WordCount or related workloads:

(1) The small executors with large memory and vcores can be the best strategy compared to other settings.

(2) Input Splits should not be considered when processing the aggregation jobs.

(3) The parallelism can improve the Spark job efficiency obviously. But the value of the parallelism should be decided based on the job itself.

6.2 K-means

For k-means or iterative workloads, Spark shows its strength with all the persistence strategies compared to MapReduce jobs. Take DISK_ONLY strategy into consideration, the original difference between Spark and MapReduce is huge when processing small datasets, but the gap narrows when the datasets grow larger. Finally, they share similar performance when the datasets reach to 150GB. Thus, we can conclude that MapReduce is good at handling larger iterative jobs while Spark are competent for all the sizes of jobs.

Regarding Spark MEMORY_ONLY strategy, the results show that the difference between DISK_ONLY strategy and MEMORY_ONLY strategy is not obvious when the dataset is small. However, it becomes more and more clear with the datasets increase. Besides, the difference between MapReduce and MEMORY_ONLY strategies is stable which means the gap is not narrow when the datasets grow. Thus, we can conclude that Spark MEMORY_ONLY strategy is better than MapReduce and Spark DISK_ONLY strategies from all the aspects.

Considering the MEMORY_AND_DISK strategy for Spark, the results show that the efficiency becomes higher with the percentage of memory increases and the gap becomes more and more obvious when the sizes of the datasets grow. This situation happens for both small and intermediate datasets. Also, we can find that the time when the biggest drop happens which represent insufficient memory appears earlier when the datasets grow. For intermediate and small datasets, the biggest drop happens when the low percentage of memory are used. Then, the drop appears at the higher percentage and finally happens at the biggest percentage. It shows that the situation of insufficient memory becomes more and more significant when the datasets grow. Thus, we can conclude that memory should be recognized as an important resource to allocate to ensure the efficiency of the Spark jobs.

The last part is all about the Spark persist strategies. The results illustrate that the utilization of memory can strongly affect the job efficiency since the difference between the worst and the best performance of MEMORY_AND_DISK strategy is clear. Also, DISK_ONLY strategy is among the best and worst performance which means that the bad memory utilization may slower than saving all the data on disks. Besides, there is an obvious gap between MEMORY_ONLY strategy and best performance of MEMORY_AND_DISK which can give us a clear idea that how strong the MEMORY_ONLY strategy can be.

In conclusion, memory is the most important resource for Spark jobs to allocate. It can not only reduce the execution time dramatically but also ensure the job efficiency. Also, Spark is a better option compare to MapReduce from all the aspects when processing the iterative tasks. Thus, we propose our suggestions that:

(1) Without considering the resource utilization, MEMORY_ONLY strategy is the best option to process the iterative tasks.

(2) When the memory is not enough to save all the intermediate data, MEMORY_AND_DISK strategy can be utilized to enhance the scalability. But the percentage of memory must be high to ensure the efficiency.

(3) Spark is better than MapReduce even with the DISK_ONLY strategy. When the memory resource is too limited, DISK_ONLY strategy can be the second choice to make the job more efficient.

6.3 TeraSort

For TeraSort or similar workloads, MapReduce presents its advantages considering the number of reduce tasks and input splits while Spark shows higher efficiency after configuring the shuffle-related parameters. For the jobs with the default settings, MapReduce is slightly faster at the beginning and the gap becomes larger when the datasets grows. Besides, when we take the number of reduce tasks into consideration, it can strongly affect the efficiency of the MapReduce jobs and the improvement rates even reach to 45%. Also, the most obvious improvement happens when processing the intermediate datasets. For Spark jobs, although the improvements rates are not as obvious as MapReduce jobs, the parameters related to the amount of reduce tasks can bring steady improvement for all the datasets especially for small datasets. In addition, the best performance for MapReduce happens when the jobs with less reduce tasks while it happens when the jobs with more reduce tasks for Spark.

Regarding to the input splits, MapReduce and Spark show different ranges and degree when tuning these parameters. For MapReduce, the best period is between 100GB and 350GB and the improvement rates are around 25%. For Spark, the period is between 100GB and 300GB which can bring 10% improvement. Also, the biggest difference happens when processing large datasets: the parameters related to MapReduce can hardly bring any improvement to the jobs while Spark parameters can provide around 10% improvement for large datasets. Besides, the optimal value of input splits for MapReduce may between small and intermediate while it should be a small value for Spark jobs.

Considering the parameter related to the shuffle process, MapReduce jobs present a compromise situation: the parameters can improve and decrease the efficiency for the jobs compared to the benchmark. Based on the situation, we can still find that setting the value slightly bigger can improve the efficiency which is about 10% for intermediate datasets. On the contrary, Spark parameters shows a steady increase for all the datasets and the improve rates are around 20%. Also, the larger values are the more efficient the job will be.

In conclusion, MapReduce is suitable for processing all the sizes of the shuffle jobs which can provide great efficiency compared to Spark. Also, the number of reduce tasks and the input splits can bring positive effects to the jobs while the reduce-side parameters can help under some certain circumstances. Besides, the best improvement rates from the amount of reduce tasks and the input splits are 45.64% and 49.54%. These values can be defined as huge improvement and strongly affect the efficiency of the jobs. Thus, based on the findings, we propose some suggestions that:

(1) Without considering any parameters settings, MapReduce should be the first choice when processing the shuffle jobs.

(2) The number of reduce tasks can strongly affect the efficiency of the shuffle jobs and thus should be considered. Also, allocating more resource to the reduce tasks with few numbers is the best strategy for MapReduce jobs.

(3) The default input splits should be set larger to improve efficiency. The value should not be too large.

(4) Reduce-side parameters should be used with caution when processing the shuffle jobs.

For Spark, although the efficiency is not as good as MapReduce, it is still a good option when processing the shuffle jobs and the difference is not obvious. Besides, all the aspects can improve the efficiency and the best improvement rates are 38.4%, 25.65% and 30.42% for the number of reduce tasks, input splits and shuffle-related jobs. Thus, the parameters can bring much improvement for the jobs either. Based on the facts, we propose our suggestions that:

(1) Spark jobs have strong flexibility when applying different parameters for shuffle jobs.

(2) The number of reduce tasks can bring positive effects for the shuffle jobs and the suggested value should be larger.

(3) Setting the input splits a little bit larger can obviously improve the job efficiency. But the value should not be too large.

(4) Shuffle-related parameters can improve the efficiency a lot and thus should be taken into consideration.

Chapter 7

Conclusion

7.1 Conclusion

In our research, we have talked about how to process the large scale of data with the existing technique. A modified approach was proposed to put different workloads into practice and deploy them on the cluster model. The new approach offers a complete process which includes data preparation, job execution and status monitoring and provides two ways to implement each workload. Furthermore, the same method can help us implement any workloads and monitor their life cycle from web UI. The major components of our experiments were comparing the performance difference between Hadoop MapReduce and Spark with three existing workloads. The results show that Spark is superior to Hadoop when processing aggregation and iterative jobs and it is more sensitive and efficient when configuring the parameters related to the workloads. On the contrary, Hadoop MapReduce shows higher efficiency when processing the shuffle jobs while the improvement rates keeps stable when facing different parameters.

A new parameter tuning method was proposed and was able to help the users optimize their job efficiency through any aspects they want. By utilizing the web UI from Ambari, we were able to set and monitor any parameters clearly.

The instruction mentioned in Chapter 3 should be followed to conduct our experiments. The users must download the Hibench Suite package and install it on their own linux system properly. Then, following the steps, they can easily implement any jobs they want and monitor them through the Ambari UI.

7.2 Future work

In this section, we are talking about the future work of our research. There are four aspects that may be consider or improved in the future: the size of the cluster, the types of our jobs, evaluation aspects and related parameters.

7.2.1 Size of the cluster

The first aspect is about the size of our cluster. Since our cluster is small which only contains 10 nodes, the ranges for aggregation and shuffle jobs are between 50GB and 500GB while the iterative jobs are between 50GB to 150GB. Although the size may big enough to simulate the situation of our daily life, we still hope to transfer our experiments to a larger cluster to see whether there are some difference. Also, we hope to enlarge the sizes our datasets to find out more interesting phenomenon.

7.2.2 Types of our jobs

The second aspect is the types of workloads. There are three types of workloads included in our research and all of them are from HiBench. Also, HiBench provides more than 15 kinds of jobs for the users to test the performance of the cluster. Thus, we hope to put more workloads into our experiments to present more aspects of our cluster.

7.2.3 Evaluation aspects

For our experiments, we utilize the execution time as the main character to present the job efficiency. Also, the execution time is from the average value after running the jobs for 5 times. Since Ambari UI provides many aspects of the jobs which includes resource utilization and the internet connection, we hope to take more aspects into consideration to find out the details of our jobs. Also, we hope to improve the accuracy of our jobs by running them for more times.

7.2.4 Parameter

Based on the documents online, we select several parameters for different jobs that most likely to affect the job efficiency. However, MapReduce and Spark provide many parameters from all the aspects of the process to enhance the job efficiency. Thus, we are going to put more parameters into our experiments to see whether they can affect the job efficiency for MapReduce and Spark.

Bibliography

[1] A. Verma, A. H. Mansuri, and N. Jain, "Big data management processing with hadoop mapreduce and spark technology: A comparison," in *Colossal Data Analysis and Networking (CDAN), Symposium on.* IEEE, 2016, pp. 1–4.

[2] B. Vaddeman, *Beginning Apache Pig: Big Data Processing Made Easy.* Apress, 2016.

[3] S. Han, W. Choi, R. Muwafiq, and Y. Nah, "Impact of memory size on bigdata processing based on hadoop and spark," in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems.* ACM, 2017, pp. 275–280.

[4] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments." in *Osdi*, vol. 8, no. 4, 2008, p. 7.

[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on.* Ieee, 2010, pp. 1–10.

[6] M. Luo and H. Yokota, "Comparing hadoop and fat-btree based access method for small file i/o applications," in *International Conference on Web-Age Information Management.* Springer, 2010, pp. 182–193.

[7] H. Yang, Z. Luan, W. Li, and D. Qian, "Mapreduce workload modeling with statistical approach," *Journal of grid computing*, vol. 10, no. 2, pp. 279–310, 2012.

[8] D. Vohra, *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools.* Apress, 2016.

[9] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel data processing with mapreduce: a survey," *AcM sIGMoD Record*, vol. 40, no. 4, pp. 11–20, 2012.

[10] T. White, *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[11] J. Venner, *Pro hadoop.* Apress, 2009.

[12] H. Herodotou, "Hadoop performance models," *arXiv preprint arXiv:1106.0940*, 2011.

[13] S. Humbetov, "Data-intensive computing with map-reduce and hadoop," in *2012 6th International Conference on Application of Information and Communication Technologies (AICT)*. IEEE, 2012, pp. 1–5.

[14] L. Gu and H. Li, "Memory or time: Performance evaluation for iterative operation on hadoop and spark," in *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*. IEEE, 2013, pp. 721–727.

[15] Y.-M. Yang, C.-D. Wang, and J.-H. Lai, "An efficient parallel topic-sensitive expert finding algorithm using spark," in *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 2016, pp. 3556–3562.

[16] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[17] Spark, "Rdd programming guide," https://spark.apache.org/docs/latest/rdd-programming-guide.html, accessed November 2, 2018.

[18] Z. Han and Y. Zhang, "Spark: A big data processing platform based on memory computing," in *2015 Seventh International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*. IEEE, 2015, pp. 172–176.

[19] P. Kannan, "Beyond hadoop mapreduce apache tez and apache spark," *San Jose State University.*, 2015.

[20] Spark, "Spark online document," https://spark.apache.org/documentation.html, accessed November 2, 2018.

[21] S. Salehian and Y. Yan, "Comparison of spark resource managers and distributed file systems," in *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)(BDCloud-SocialCom-SustainCom)*. IEEE, 2016, pp. 567–572.

[22] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning spark: lightning-fast big data analysis*. " O'Reilly Media, Inc.", 2015.

[23] M. Frampton, *Mastering apache spark*. Packt Publishing Ltd, 2015.

[24] C. Vazquez, R. Krishnan, and E. John, "Cloud computing benchmarking: A survey," in *Proceedings of the International Conference on Grid Computing and Applications*

(GCA). The Steering Committee of The World Congress in Computer Science, Computer . . . , 2014, p. 1.

[25] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*. IEEE, 2010, pp. 41–51.

[26] L. Mashayekhy, M. M. Nejad, D. Grosu, Q. Zhang, and W. Shi, "Energy-aware scheduling of mapreduce jobs for big data applications," *IEEE transactions on Parallel and distributed systems*, vol. 26, no. 10, pp. 2720–2733, 2015.

[27] C. z. Guo, "Hibench suite," https://github.com/intel-hadoop/HiBench, 2018.

[28] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[29] Nutch, "Nutch online document," http://nutch.apache.org/.

[30] T. Ivanov, R. Niemann, S. Izberovic, M. Rosselli, K. Tolle, and R. V. Zicari, "Performance evaluation of enterprise big data platforms with hibench," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, vol. 2. IEEE, 2015, pp. 120–127.

[31] Y. Samadi, M. Zbakh, and C. Tadonki, "Comparative study between hadoop and spark based on hibench benchmarks," in *Cloud Computing Technologies and Applications (CloudTech), 2016 2nd International Conference on*. IEEE, 2016, pp. 267–275.

[32] A. Song, Z. Wu, X. Ma, and J. Luo, "Cat: A cost-aware translator for sql-query workflow to mapreduce jobflow," *Data & Knowledge Engineering*, vol. 102, pp. 42–56, 2016.

[33] L.-H. Xiang, L. Miao, D.-F. Zhang, and F.-P. Chen, "Benefit of compression in hadoop: A case study of improving io performance on hadoop," in *Proceedings of the 6th International Asia Conference on Industrial Engineering and Management Innovation*. Springer, 2016, pp. 879–890.

[34] Dong, "Hadoop-terasort-analysis," https://spark.apache.org/docs/latest/cluster-overview.html, march 20, 2011.

[35] Kuaibao, "The secret of spark," https://kuaibao.qq.com/s/20180412A1MULR00?refer=spider, april 12, 2018.

[36] O. O'Malley, "Terabyte sort on apache hadoop," *Yahoo, available online at: http://sortbenchmark. org/Yahoo-Hadoop. pdf,(May)*, pp. 1–3, 2008.

[37] S. Li, S. Supittayapornpong, M. A. Maddah-Ali, and S. Avestimehr, "Coded terasort," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE, 2017, pp. 389–398.

[38] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker, "Prefix hash tree: An indexing data structure over distributed hash tables," in *Proceedings of the 23rd ACM symposium on principles of distributed computing*, vol. 37, 2004.

[39] K.-H. Li, "Reservoir-sampling algorithms of time complexity o (n (1+ log (n/n))),"
ACM Transactions on Mathematical Software (TOMS), vol. 20, no. 4, pp. 481–493, 1994.

[40] R. Barga, V. Fontama, W. H. Tok, and L. Cabrera-Cordon, *Predictive analytics with Microsoft Azure machine learning*. Springer, 2015.

[41] F. zhe, *Mahour Algorithm Analysis and Case practice[M]*. Beijing:China Machine Press, 2014.

[42] S. Gopalani and R. Arora, "Comparing apache spark and map reduce with performance analysis using k-means," *International journal of computer applications*, vol. 113, no. 1, 2015.

[43] R. M. Esteves, R. Pais, and C. Rong, "K-means clustering in the cloud–a mahout test," in *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*. IEEE, 2011, pp. 514–519.

[44] K. D. Garcia and M. C. Naldi, "Multiple parallel mapreduce k-means clustering with validation and selection," in *Intelligent Systems (BRACIS), 2014 Brazilian Conference on*. IEEE, 2014, pp. 432–437.

[45] H. shan, "The implemenation of k-means in mahout," https://www.cnblogs.com/biyeymyhjob/archive/2012/07/20/2599544.html, july 20, 2012.

[46] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035.

[47] I. Kusuma, M. A. Ma'sum, N. Habibie, W. Jatmiko, and H. Suhartanto, "Design of intelligent k-means based on spark for big data clustering," in *Big Data and Information Security (IWBIS), International Workshop on*. IEEE, 2016, pp. 89–96.

[48] A. Holmes, *Hadoop in practice*. Manning Publications Co., 2012.

[49] Hadoop, "Apache hadoop 2.9.2," https://hadoop.apache.org/docs/current/, november 13, 2018.

[50] H. Karau and R. Warren, *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark.* " O'Reilly Media, Inc.", 2017.

[51] S. Wadkar, M. Siddalingaiah, and J. Venner, *Pro Apache Hadoop.* Springer, 2014.

[52] M. Reza, B. Tripathy, H. Ranjan, and G. A. Kumar, "Study and analysis of hadoop cluster optimization based on configuration properties," in *2017 Innovations in Power and Advanced Computing Technologies (i-PACT).* IEEE, 2017, pp. 1–4.

[53] Ambari, "Apache ambari," https://ambari.apache.org/, november 16, 2018.

[54] P. Raj, "The hadoop ecosystem technologies and tools," in *Advances in Computers.* Elsevier, 2018, vol. 109, pp. 279–320.

Appendix A

Tables

The tables in the appendix are the results of our experiments in details. Table 8.1 and 8.2 represent the results of WordCount in MapReduce and Spark. Table 8.3 represents the results of k−means in both while Table 8.4 and 8.5 illustrates the results of TeraSort. Since we did each experiment for five times, the red numbers represent the average execution time followed by the five specific execution time which colors are black. Also, we calculate the standard deviation with STDEVA function from excel and put it behind the average execution time. Besides, the outlier which means excess 20% of the average execution time was canceled by the cross sign.

Hadoop	50G	100G	150G	200G	250G	300G	350G	400G	450G	500G
Resource utilization										
7G 1vcore:map 14G1cvore:reduce (default)	212 ±32	407 ±41	600 ±63	790 ±67	979 ±76	1170 ±81	1364 ±59	1549 ±69	1769 ±205	1934 ±125
	248	479	705	902	925	1195	1325	1646	2065	2065
	245	401	600	766	932	1027	1322	1552	1670	1970
	185	384	547	801	936	1203	1318	1454	1870	2007
	194	382	591	741	1104	1222	1415	1527	1529	1745
	188	389	557	740	998	1203	1440	1566	1711	1883
4G 1cvore: map 8G1vcore:reduce	192 ±11	369 ±18	538 ±14	716 ±11	887 ±21	1054 ±13	1226 ±36	1400 ±99	1587 ±27	1769 ±75
	189	370	537	710	889	1061	1230	1407	1562	1749
	194	369	539	715	887	1060	1229	1411	1560	1748
	177	368	542	725	901	1052	1170	1502	1595	1828
	208	394	555	702	906	1033	1230	1237	1626	1855
	192	344	517	728	852	1064	1271	1443	1592	1665
5G 1cvore: map 10G 1vcore:reduce	197 ±8	378 ±15	559 ±24	717 ±40	898 ±54	1065 ±66	1242 ±84	1445 ±83	1625 ±84	1805 ±90
	199	395	573	691	954	1130	1335	1497	1665	1861
	200	385	~~715~~	760	947	1140	1217	1318	1700	1876
	207	386	580	756	841	1037	1117	1413	1688	1693
	191	358	526	709	846	995	1244	1531	1557	1874
	188	366	557	669	902	1023	1297	1466	1515	1721
6G 1vcore: map 12G 1vcore:reduce	201 ±29	392 ±43	581 ±48	755 ±62	936 ±67	1114 ±93	1307 ±109	1505 ±87	1665 ±131	1865 ±138
	218	452	619	831	1045	1234	1446	1636	1848	2050
	219	421	625	737	940	1138	1247	1439	1543	1953
	156	362	532	731	931	975	1352	1441	1631	1837
	188	348	602	675	888	1117	1159	1457	1748	1701
	224	377	527	801	876	1106	1331	1552	1555	1784
8G 2vcores: map 16G 2vcores:reduce	235 ±45	444 ±84	653 ±91	867 ±60	1078 ±84	1464 ±147	1507 ±120	1707 ±147	1907 ±123	2115 ±115

	300	575	778	958	1149	1434	1523	1706	1983	2086
	202	478	678	862	1048	1535	1632	1905	2080	2176
	263	394	661	870	945	1223	1601	~~2308~~	1826	2283
	209	406	526	789	1137	1595	1337	1552	1777	2029
	201	367	622	856	1111	1533	1442	1665	1869	2001
9G 2vcores: map	256	506	737	981	1231	1460	1701	1940	2192	2417
18G 2vcores:reduce	±40	±65	±87	±112	±51	±56	±32	±113	±94	±170
	304	572	871	1157	1250	1529	1716	2000	2291	2687
	279	574	774	959	1258	1418	1728	1901	2183	2482
	207	493	685	864	1148	1444	1729	2102	2087	2281
	224	436	656	914	1279	1402	1665	1809	2285	2334
	266	455	699	1011	1220	1507	1667	1888	2114	2301
10G 2vcores: map	259	507	744	983	1225	1470	1700	1933	2186	2418
20G 2vcores:reduce	±47	±56	±89	±86	±30	±54	±59	±127	±123	±48
	225	603	881	881	1212	1503	1788	2118	2118	2385
	307	492	695	1015	1210	1499	1691	1888	2088	2381
	312	502	718	918	1192	1525	1697	1885	2085	2405
	235	471	649	1100	1245	1410	1623	1782	2332	2419
	216	467	777	1001	1266	1413	1701	1992	2307	2500
Input Splits:										
256M	205	364	541	712	896	1047	1223	1380	1537	1720
	±36	±49	±66	±83	±64	±43	±58	±116	±90	±32
	233	402	601	630	852	1054	1252	1402	1660	1701
	228	323	557	730	965	1052	1261	1442	1593	1693
	167	411	599	621	850	1032	1258	1227	1473	1706
	233	~~466~~	498	777	845	989	1123	1305	1438	1772
	164	320	450	802	968	1108	1221	1524	1521	1728
512M	168	360	531	690	824	1006	1140	1312	1448	1631
	±22	±49	±63	±54	±43	±22	±50	±77	±94	±47
	198	345	567	698	802	996	1172	1364	1574	1656
	183	426	601	702	806	991	1173	1368	1474	1655
	160	387	489	598	797	1000	1178	1371	1477	1666
	155	344	552	741	814	999	1065	1213	1333	1627
	144	298	446	711	901	1044	1112	1244	1382	1551
1024M	221	370	525	688	841	1022	1132	1279	1429	1572
	±47	±46	±47	±55	±45	±27	±51	±123	±56	±48
	267	418	569	603	840	998	1157	1383	1444	1533
	274	317	582	692	831	1018	1195	1385	1448	1539
	168	326	477	672	839	994	1148	~~1650~~	1473	1550
	197	385	501	744	784	1045	1072	1147	1331	1648
	199	404	496	729	911	1055	1088	1201	1449	1590
Map-Side parameters										
default(2047,100)										
i/o.sort.mb=1024	202	394	574	753	946	1130	1314	1486	1674	1853
i/o.sort.factor=50	±55	±49	±47	±69	±30	±35	±62	±56	±89	±24
	238	471	612	694	959	1159	1304	1451	1604	1845
	141	362	501	695	925	1155	1412	1415	1612	1832
	144	367	600	732	915	1153	1313	1482	1613	1831
	255	415	552	857	990	1096	1240	1554	1759	1880
	232	355	605	787	941	1087	1301	1528	1782	1877
(1536,75)	206	400	592	771	963	1155	1326	1531	1713	1891
	±35	±46	±23	±65	±47	±33	±91	±34	±92	±47
	240	468	598	798	924	1165	1393	1558	1610	1843
	248	426	605	705	930	1164	1234	1563	1719	1846
	187	375	599	699	933	1097	1392	1501	1632	1933
	182	358	551	841	1011	1167	1391	1488	1826	1890
	173	373	607	812	1017	1182	1220	1545	1778	1943
(512,25)	200	372	560	735	922	1091	1266	1461	1608	1769
	±34	±43	±36	±57	±29	±51	±73	±64	±55	±52
	230	345	602	770	921	1128	1343	1458	1603	1780
	194	447	569	676	901	1033	1348	1563	1701	1821
	162	360	533	679	973	1059	1202	1396	1588	1782
	240	344	582	806	907	1077	1216	1466	1556	1682
	174	364	514	744	908	1158	1221	1422	1592	1780

Table A.1: The results of Wordcount in MapReduce

Spark	50G	100G	150G	200G	250G	300G	350G	400G	450G	500G
Resource utilization										
50 executors 8G memory 4 vcores	56 ±8	83 ±13	120 ±16	145 ±24	189 ±14	201 ±29	243 ±30	266 ±21	285 ±27	333 ±17
	66	~~105~~	124	170	203	222	290	281	304	347
	59	95	135	166	176	224	248	283	266	345
	~~75~~	93	102	124	197	167	218	230	308	342
	52	70	134	148	171	171	244	265	247	308
	47	74	105	117	198	221	215	271	300	323
60 7 4	56 ±11	94 ±5	120 ±18	166 ±12	200 ±8	220 ±9	276 ±23	297 ±14	316 ±16	371 ±35
	66	102	120	167	198	225	250	293	306	352
	69	93	137	155	192	223	253	288	298	415
	45	95	103	153	202	220	292	282	329	338
	48	87	138	177	212	204	301	318	312	349
	52	93	102	178	196	228	284	304	335	401
70 6 3	58 ±10	93 ±11	114 ±17	169 ±10	213 ±20	254 ±18	286 ±5	327 ±38	355 ±41	410 ±66
	~~74~~	94	137	162	205	267	286	341	305	347
	68	102	113	163	202	228	283	312	367	481
	52	104	122	172	197	246	287	387	321	345
	48	78	94	185	214	274	294	299	405	401
	64	87	104	163	247	255	280	296	377	476
80 5 3	61 ±8	97 ±15	127 ±17	170 ±7	209 ±11	249 ±21	272 ±16	318 ±33	328 ±37	379 ±25
	71	99	134	167	212	236	263	306	308	366
	50	100	148	165	205	225	266	312	295	405
	61	84	130	180	203	248	263	291	344	356
	57	120	101	174	198	281	267	305	307	360
	66	82	122	164	227	255	301	376	386	408
90 4 2	63 ±8	95 ±14	127 ±14	162 ±7	184 ±22	218 ±11	251 ±15	290 ±23	300 ±23	353 ±14
	56	112	128	171	198	233	254	289	325	364
	71	79	118	165	201	226	253	326	308	351
	72	96	137	157	156	210	273	268	315	344
	57	83	108	164	200	214	232	296	274	336
	59	105	144	153	165	207	243	271	278	370
100 4 2	60 ±11	97 ±14	132 ±5	161 ±14	193 ±12	216 ±14	247 ±15	287 ±8	317 ±14	341 ±33
	74	104	130	174	200	233	268	289	303	355
	53	116	134	151	177	205	243	294	312	380
	45	87	139	178	204	230	237	287	307	302
	67	81	126	149	200	203	256	292	338	310
	61	97	131	153	184	209	231	273	325	358
Input splits										
256M	70 ±8	119 ±12	146 ±18	202 ±34	260 ±36	310 ±20	341 ±33	365 ±35	404 ±26	454 ±39
	78	138	165	230	268	~~416~~	324	330	396	467
	68	120	141	246	299	331	369	408	438	~~606~~
	62	119	157	190	225	301	304	329	424	435
	79	107	117	166	287	322	383	386	385	502
	63	111	150	178	221	286	325	372	377	412
512M	78 ±9	115 ±8	187 ±21	216 ±22	291 ±33	305 ±27	340 ±41	353 ±31	383 ±33	464 ±37
	80	~~153~~	158	236	336	276	344	314	401	456
	86	107	200	244	247	304	407	327	353	513
	87	125	210	200	277	281	316	367	341	490
	66	109	175	197	293	328	298	389	411	428
	71	119	192	203	302	336	335	368	409	433
1024M	90 ±16	160 ±16	203 ±24	249 ±20	277 ±37	327 ±27	355 ±30	363 ±18	399 ±28	480 ±31
	94	188	192	251	311	296	336	337	429	480
	77	156	222	242	238	316	357	361	360	440
	117	151	234	260	251	360	321	360	417	490
	83	158	190	219	321	312	399	386	407	465
	79	147	177	273	264	351	362	371	382	525
Parallelism										
100	54 ±9	79 ±12	113 ±13	134 ±15	176 ±9	199 ±17	230 ±15	277 ±14	287 ±20	309 ±25

	59	95	116	151	163	212	233	269	290	337
	66	69	132	137	183	215	240	274	280	336
	45	74	105	126	186	192	236	273	318	286
	52	89	99	112	177	174	238	301	263	297
	48	68	113	144	171	202	203	268	284	289
200	60 ±8	84 ±9	108 ±9	151 ±10	183 ±23	201 ±21	234 ±16	265 ±25	286 ±9	328 ±14
	70	90	103	153	213	230	243	272	289	343
	55	84	111	156	189	213	243	302	292	335
	67	72	121	139	151	174	207	237	296	319
	52	96	98	163	188	195	246	263	274	335
	56	78	107	144	174	193	231	251	279	308
300	60 ±7	93 ±5	108 ±13	140 ±10	189 ±2	195 ±26	239 ±11	275 ±9	295 ±19	347 ±12
	71	95	130	145	189	220	257	269	286	349
	52	94	108	135	189	195	229	282	318	345
	58	99	97	152	187	222	242	285	286	351
	63	90	101	127	192	167	230	263	311	329
	56	87	104	141	188	171	237	276	274	361
400	62 ±6	88 ±13	114 ±9	146 ±16	169 ±13	207 ±19	248 ±13	284 ±14	292 ±13	326 ±18
	65	99	126	159	187	236	244	274	295	352
	59	100	110	127	160	215	245	285	296	324
	71	73	107	148	171	192	242	272	297	335
	57	92	122	132	154	203	271	308	269	306
	58	76	105	164	173	189	238	281	303	313
500	60 ±9	86 ±12	119 ±7	144 ±17	173 ±22	205 ±12	240 ±14	278 ±5	286 ±13	342 ±13
	70	90	130	124	179	218	244	281	288	350
	69	77	112	155	209	212	250	277	300	343
	52	104	120	167	160	208	239	283	296	330
	55	85	119	143	152	189	250	269	279	359
	54	74	114	131	165	198	217	280	267	328

Table A.2: The results of Wordcount in Spark

	50G	75G	100G	125G	150G
MapReduce	591 ±16	835 ±23	1119 ±36	1337 ±32	1630 ±22
	607	860	1180	1382	1662
	587	847	1098	1326	1637
	593	822	1121	1357	1607
	602	844	1102	1311	1611
	566	802	1094	1309	1633
DISK_ONLY	194 ±19	267 ±28	646 ±25	944 ±62	1554 ±22
	222	~~338~~	666	836	1579
	177	298	621	974	1545
	179	235	678	992	1566
	188	255	639	966	1521
	204	280	626	952	1559
MEMORY_ONLY	73 ±13	117 ±14	140 ±11	198 ±18	561 ±44
	74	136	155	215	523
	67	104	124	187	615
	95	126	144	199	580
	69	117	137	174	510
	60	102	140	215	577
(MEMORY_AND_DISK)					
50%	166 ±17	262 ±29	725 ±28	1528 ±177	1888 ±187
	186	232	717	1248	1672
	143	250	766	1496	~~2465~~
	177	244	~~977~~	1544	~~3462~~
	168	299	711	1650	1995
	156	285	706	1702	1997
60%	138 ±14	240 ±22	600 ±16	1140 ±58	1660 ±46

	121	243	597	1057	1710
	128	266	625	1142	1619
	155	251	606	1113	1659
	145	208	588	1199	1702
	141	232	584	1189	1610
70%	125 ±13	221 ±15	572 ±20	1020 ±135	1607 ±261
	118	227	581	975	1323
	137	214	550	~~1289~~	~~2183~~
	142	198	603	1220	1885
	115	234	553	923	1762
	113	232	573	962	1458
80%	117 ±8	202 ±16	454 ±46	836 ±64	1229 ±48
	127	226	458	834	1299
	122	207	522	926	~~1772~~
	105	181	468	868	1189
	114	199	412	778	1207
	117	197	410	774	1221
90%	111 ±9	186 ±6	387 ±58	693 ±25	901 ±122
	116	189	~~481~~	702	764
	101	188	369	688	~~660~~
	124	193	~~531~~	723	996
	110	177	452	697	833
	104	183	340	655	1011
30 executors with 8G memory and 2G memory overhead					

Table A.3: The results of k−means

Hadoop	50G	100G	150G	200G	250G	300G	350G	400G	450G	500G
Resource utilization										
14G 1vcore:reduce 30 :reduce number (default)	190 ±26	346 ±20	619 ±34	987 ±88	1076 ±100	1779 ±165	1919 ±57	2011 ±127	2482 ±195	2779 ±168
	210	364	630	1062	986	1800	1946	1902	2249	2600
	223	350	655	1100	953	2020	1969	1860	2310	2704
	166	362	642	904	1130	1795	1907	2069	2692	2820
	186	315	572	943	1183	1564	1948	2167	2631	2992
	165	339	596	926	1128	1716	1825	2057	2528	~~3496~~
10G 1vcores: reduce 40:reduce number	230 ±30	422 ±78	847 ±37	1268 ±55	1471 ±30	1786 ±114	2186 ±91	2678 ±140	2886 ±231	2914 ±364
	261	509	870	1241	1468	1617	2270	2809	2611	2711
	246	501	861	1223	1421	1765	2053	2652	2715	2505
	193	370	887	1292	1482	1821	2197	2478	2892	3392
	220	338	799	1353	1501	1794	2143	2815	3027	2777
	174	392	818	1231	1483	1933	2267	2636	3185	3185
12G 1vcores: reduce 35:reduce number	221 ±36	405 ±22	745 ±39	1229 ±123	1315 ±53	1741 ±55	2075 ±36	2265 ±233	2503 ±71	2795 ±223
	255	377	729	1339	1339	1784	2101	2414	2466	2924
	174	424	719	1168	1361	1777	2121	2530	2444	3075
	259	430	813	1381	1299	1744	2073	2337	2622	2834
	211	394	722	1104	1230	1752	2043	2040	2510	2540
	206	400	742	1153	1346	1648	2037	2004	2473	2602
16G 2vcores: reduce 25:reduce number	214 ±9	374 ±55	582 ±20	825 ±21	1079 ±77	1350 ±130	1611 ±148	2024 ±56	2461 ±70	2723 ±299
	218	408	583	807	1030	1272	1515	2067	2339	3043
	223	419	595	823	1017	1253	1530	2029	2494	2876
	218	414	580	805	1023	1248	1474	2079	2468	2399
	210	315	602	835	1145	1445	1811	2008	2500	2408
	201	314	550	855	1180	1532	1725	1937	2504	2889
18G 2cvore: reduce 20:reduce number	205 ±15	327 ±24	556 ±49	813 ±44	1023 ±56	1223 ±170	1457 ±222	1734 ±58	2058 ±72	2412 ±75
	221	328	602	858	1096	1320	1636	1634	2062	2520
	218	343	575	846	1045	1343	1650	1756	2067	2439
	201	357	594	790	946	1037	1227	1737	2171	2350
	185	304	489	749	998	1040	1207	1779	1990	2335

	200	303	520	822	1030	1375	1565	1764	2000	2416
Input Splits:										
256M	143 ±6	316 ±16	486 ±44	705 ±46	930 ±101	1217 ±70	1680 ±82	1946 ±241	2393 ±95	2705 ±162
	147	295	457	679	1074	1328	1607	2139	2420	2938
	134	303	454	663	~~1194~~	1233	1777	~~2463~~	2530	2779
	141	332	461	713	878	1178	1587	1762	2411	2700
	150	328	558	690	922	1202	1740	2168	2298	2560
	143	322	500	780	846	1144	1689	1715	2306	2548
512M	187 ±35	269 ±42	466 ±24	660 ±90	903 ±87	1255 ±52	1514 ±118	2170 ±85	2561 ±224	2837 ±175
	191	316	458	637	~~1140~~	1281	1674	2188	2297	3081
	142	251	435	545	994	1165	1343	2168	2746	2896
	168	310	463	620	961	1284	1532	2302	2830	2847
	234	220	474	771	824	1290	1521	2100	2520	2751
	200	248	500	727	833	1255	1500	2092	2412	2610
1024M	182 ±19	319 ±33	540 ±61	705 ±64	1042 ±66	1304 ±45	1560 ±139	1969 ±86	2780 ±143	2993 ±217
	~~245~~	340	640	744	1004	1320	1430	1867	2977	2977
	192	338	486	713	965	1367	1463	2087	2802	2802
	204	349	549	616	1020	1287	1690	2011	2622	2839
	169	276	520	671	1120	1301	1487	1972	2660	2996
	163	292	505	781	1101	1245	1730	1908	2839	3351
Reduce-Side parameters										
default(100,30) mapreduce.reduce. shuffle.parallelcopies =60 mapreduce.task.io. sort.factor =200	184 ±24	415 ±58	612 ±41	894 ±122	1169 ±67	1719 ±64	1911 ±206	2394 ±169	3032 ±260	3429 ±88
	219	342	568	817	1087	1653	1772	2585	2847	3439
	159	462	650	780	1238	1817	2208	2328	3327	3302
	171	~~522~~	660	1050	1110	1681	2039	2556	2809	3414
	195	395	582	999	1190	1743	1716	2302	3303	3440
	176	461	600	824	1220	1701	1820	2199	2874	3550
(150,45)	176 ±6	372 ±24	562 ±62	813 ±115	1175 ±64	1501 ±126	1998 ±45	2267 ±103	2880 ±85	3594 ±237
	186	332	644	965	1160	1460	2070	2216	2807	3204
	171	380	612	820	1108	1375	1947	2161	2862	3567
	175	379	522	877	1252	1408	1994	2334	2890	3695
	171	374	529	721	1230	1592	2000	2211	3020	3673
	177	395	503	682	1125	1670	1979	2413	2821	3831
(75,20)	185 ±19	350 ±15	608 ±59	963 ±137	1133 ±52	1482 ±106	1996 ±114	2273 ±221	2785 ±136	3109 ±148
	168	370	577	844	1168	1465	1934	2555	2711	3119
	163	358	529	905	1131	1641	2071	2365	2762	3349
	187	349	662	1103	1194	1396	2145	1967	3016	3092
	206	334	601	845	1112	1525	1855	2312	2770	3027
	201	339	671	1118	1060	1383	1975	2166	2666	2958
(50,15)	189 ±12	358 ±26	644 ±97	871 ±61	1242 ±134	1619 ±89	1901 ±144	2495 ±72	2718 ±160	2884 ±379
	187	345	572	842	1067	1712	1987	2439	2564	3305
	170	344	572	804	1253	1479	1972	2449	2871	2714
	192	337	770	931	1177	1606	2037	2595	2857	3279
	200	362	577	838	1430	1676	1820	2547	2763	2511
	196	402	729	940	1283	1622	1689	2445	2535	2611

Table A.4: The results of Terasort in MapReduce

Spark	50G	100G	150G	200G	250G	300G	350G	400G	450G	500G
Resource utilization (70 executors 6G memory 3 vcores)										
Reducer number:20	287 ±15	584 ±54	961 ±118	1261 ±139	1712 ±51	2211 ±191	2584 ±84	3113 ±136	3258 ±185	3656 ±165
	303	527	1011	1450	1736	2267	2510	2987	2969	3603

	282	564	815	1116	1657	2497	2655	2949	3179	3874
	290	619	881	~~1610~~	~~2167~~	2011	2483	3243	3384	3465
	265	550	1120	1256	1684	2213	2667	3220	3348	3772
	295	660	978	1222	1771	2067	2605	3166	3410	3566
25	253 ±38	505 ±36	899 ±68	1183 ±84	1588 ±177	2043 ±148	2467 ±92	2651 ±82	2869 ±131	3096 ±115
	283	549	871	1261	1366	1938	2318	2573	3097	3284
	210	532	802	1112	1565	2102	2506	2724	2821	3077
	212	460	918	1082	1477	1839	2558	2552	2843	2983
	277	481	918	1195	1757	2180	2448	2696	2819	3031
	283	503	986	1265	1775	2156	2505	2710	2765	3105
30	239 ±11	496 ±43	857 ±66	1093 ±111	1444 ±126	1826 ±79	2067 ±73	2445 ±184	2794 ±71	3042 ±192
	234	475	764	1180	1269	1811	2158	2343	2717	3138
	246	473	931	1222	1355	1859	2049	2771	2768	3097
	225	466	869	1089	1537	1859	2127	2349	2856	3226
	236	570	820	950	1502	1697	2004	2411	2749	2724
	254	496	901	1024	1557	1904	1997	2351	2880	3025
35	231 ±16	435 ±58	677 ±68	1051 ±89	1239 ±87	1764 ±77	1947 ±215	2019 ±115	2529 ±83	2839 ±81
	240	500	743	1046	1099	1633	2236	2054	2563	2912
	212	436	603	912	1283	1787	1812	2165	2656	2938
	~~202~~	367	697	1086	1216	1802	2117	2040	2501	2810
	247	484	734	1155	1322	1766	1750	1987	2449	2764
	225	388	608	1056	1275	1832	1820	1849	2476	2771
40	232 ±26	425 ±69	619 ±71	984 ±104	1222 ±156	1531 ±107	1698 ±193	1967 ±155	2475 ±82	2750 ±118
	236	458	699	930	1387	1543	1887	2159	2480	2599
	195	481	555	1093	1359	1520	1643	2090	2546	2790
	232	351	658	901	1024	1703	1521	1799	2417	2892
	269	485	649	896	1229	1468	1917	1945	2371	2807
	228	350	534	1100	1111	1421	1522	1842	2561	2662
Input splits (reducer number:30)										
256M	266 ±24	457 ±32	682 ±81	919 ±61	1371 ±44	1612 ±191	2052 ±136	2147 ±158	2507 ±221	2928 ±127
	~~201~~	507	717	~~1130~~	1366	1891	2210	2060	2663	3112
	278	421	766	913	1398	1708	2117	2352	2780	3002
	234	456	606	988	1296	1558	2105	2280	2456	2848
	264	443	735	935	1392	1407	1872	2003	2422	2798
	288	458	586	840	1403	1496	1956	2040	2214	2880
512M	252 ±20	491 ±33	792 ±75	1101 ±61	1451 ±162	1845 ±133	2072 ±152	2188 ±93	2806 ±220	3228 ±127
	255	518	723	1114	1403	1954	2285	2332	2713	3178
	233	444	742	1011	1233	1967	2105	2113	2475	3122
	284	503	~~1050~~	1075	1409	1639	1863	2194	2904	3140
	242	470	816	1172	1553	1810	2076	2100	2888	3432
	246	520	887	1133	1657	1855	2031	2201	3050	3268
1024M	297 ±14	532 ±35	848 ±88	1215 ±135	1465 ±117	1895 ±136	2309 ±122	2695 ±248	2989 ±297	3151 ±283
	279	482	978	1083	1392	1831	2278	2912	3193	3371
	289	542	789	1238	1525	1729	2225	2740	3374	3389
	296	569	753	1332	1323	1853	2210	2949	2966	3310
	307	555	834	1067	1624	2070	2320	2436	2735	2846
	314	512	886	1355	1461	1992	2512	2438	2677	2839
Shuffle parameters										
spark.shuffle. file.buffer=16k spark.reducer. maxSizeInFlight=24M	233 ±13	501 ±33	849 ±115	1135 ±120	1509 ±84	1610 ±161	2131 ±47	2486 ±203	3007 ±119	3080 ±147
	247	478	729	1036	1517	1819	2102	2795	2878	3297
	243	470	~~1070~~	1002	1446	1668	2112	2452	2957	3017
	235	503	798	1287	1618	1649	2158	2258	3004	3004
	219	500	998	1216	1555	1392	2083	2548	2995	3155
	221	554	871	1134	1409	1522	2200	2377	3201	2927
(32k,48M)	212 ±37	492 ±41	750 ±87	1084 ±120	1496 ±102	1565 ±112	2006 ±284	2446 ±245	2621 ±140	2796 ±217
	232	508	698	1213	1467	1694	1963	2161	2749	3159
	248	522	730	968	1410	1413	2285	2299	2610	2839

	171	420	885	1214	1411	1655	2307	2796	2772	2662
	236	507	661	1003	1650	1531	1673	2564	2523	2675
	173	503	776	1022	1542	1532	1802	2410	2451	2645
(64k,96M)	238 ±31	471 ±58	744 ±32	920 ±89	1289 ±82	1439 ±190	1909 ±100	2138 ±180	2434 ±92	2731 ±144
	198	536	730	907	1323	1604	1986	2360	2443	2742
	214	474	785	1000	1179	1682	1762	2184	2547	2881
	270	380	718	1018	1366	1275	1882	2238	2348	2840
	244	462	772	808	1352	1289	2019	1935	2497	2518
	264	503	715	867	1225	1345	1896	1973	2335	2674
(128k,192M)	193 ±35	444 ±29	673 ±79	947 ±123	1191 ±101	1400 ±65	1675 ±135	1990 ±210	2410 ±173	2634 ±260
	232	422	727	1087	1271	1491	1672	2062	2219	2335
	229	481	617	1007	1283	1371	1901	2240	2355	2702
	156	465	786	992	1035	1390	1635	2098	2302	2511
	179	441	613	775	1155	1318	1623	1818	2530	2588
	169	411	622	874	1211	1430	1544	1732	2644	3034
Replication of Spark jobs =1										

Table A.5: The results of Terasort for Spark